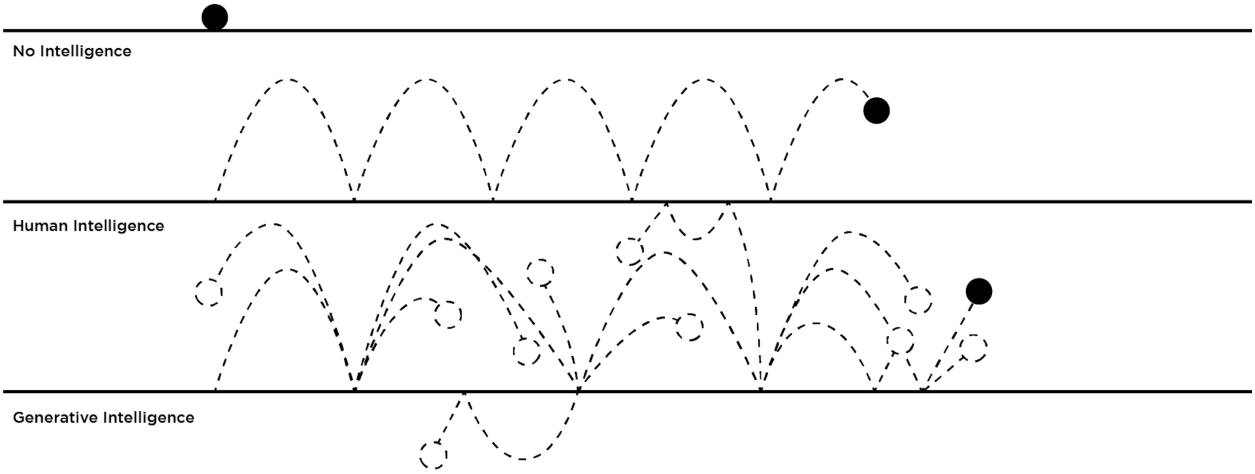
INFINITE RECURSION; RETHINKING THINKING: AUTOMATING THE AI RESEARCHER WITH GENERATIVE ALGORITHMS

Jepson Taylor
jepson@veox.ai
(former Chief AI Strategist @Dataiku & DataRobot)

Benjamin Taylor
(former quant, 14 AI patents, Zeff.ai co-founder, former Chief Data Scientist HireVue (Sequoia))

February 19, 2025



ABSTRACT

This paper introduces a comprehensive framework for *automating AI research* through the recursive use of generative algorithms. By leveraging powerful language models and evolutionary principles, we demonstrate how one can (1) generate new, highly intricate optimization problems, and (2) discover entirely novel optimization algorithms that often outperform long-established methods. We begin with a detailed Hello World exposition, showcasing a family of both *classic* and *biomechanically inspired* benchmark functions alongside a unified approach for scoring landscape complexity. These benchmarks highlight key optimization challenges, including multimodality, anisotropy, steep gradients, plateau regions, and fractal-like boundaries. We then illustrate how careful prompt engineering and iterative feedback loops enable large language models to propose genuinely sophisticated functions that score highly on our *log difficulty* metric—a measure capturing ruggedness, oscillatory behavior, and non-separability. We then extend the work to show novel competitive generative optimization algorithms and conclude with the same approach being applied to a new category of generative machine learning, where LLMs create on-demand algorithms tuned to the dataset.

1 Introduction

The field of artificial intelligence has seen remarkable progress in recent years, driven by advances in deep learning, reinforcement learning, and more recently, generative models. However, the process of AI research itself remains largely manual, requiring significant human intervention in model design, architecture selection, and hyperparameter optimization. If you look at the history of machine learning algorithms and when they were invented it is a sparse timeline. Some even argue the last significant breakthrough in the algorithm space was the transformer in 2017. The recent break-throughs in LLMs have been more of engineering marvels, rather than mathematical.

This paper introduces a novel approach to automating the AI research process through the application of generative algorithms in a recursive framework. We explore how meta-learning and self-improvement can be achieved through carefully designed recursive systems that optimize their own architecture and learning processes. We also demonstrate the importance of using adaptive systems that are capable of modifying prompts in-realtime to maximize an outcome.

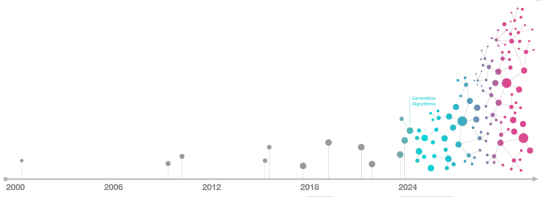


Figure 1: The timeline for meaningful human-invented algorithms in machine learning is sparse, with the capabilities of generative machine learning that is about to change quickly.

Our work builds upon several key insights:

- The potential for recursive self-improvement in AI systems
- The role of generative algorithms in automating complex design decisions
- The importance of meta-learning in creating more efficient AI research processes
- The LLMs have enough structure knowledge to help drive autonomous innovation more than we realize.

We present both theoretical foundations and practical implementations of our approach, demonstrating its effectiveness across various domains and problem types. We show evidence of novel algorithm creation in cost functions, optimizers, and in machine learning, the very foundation.

2 History of Algorithms

The evolution of algorithms has been fundamental to the development of computer science and artificial intelligence. From early mathematical procedures to modern computational methods, algorithms have shaped our approach to problem-solving and automation. In this section, we provide a historical overview of algorithmic development, highlighting key milestones that have advanced both theoretical understanding and practical innovations.

2.1 Classical Algorithms

The roots of algorithmic thinking can be traced back to antiquity. One of the earliest known examples is *Euclid's algorithm* for computing the greatest common divisor [4], which demonstrates that even simple procedures can have enduring computational value. Centuries later, the work of the 9th-century Persian scholar Al-Khwarizmi laid the groundwork for systematic methods in arithmetic and algebra; indeed, his name is the origin of the term "algorithm" [10]. Over time, classical algorithms evolved to address tasks such as sorting and searching, and their principles continue to underpin modern algorithm design [11].

2.2 Modern Developments

The mid-20th century ushered in a revolution in algorithm design with the advent of electronic computers. Pioneering work by Turing [30] established the theoretical limits of computation and laid the foundation for computer science. During this era, efficient algorithms for sorting (e.g., QuickSort, MergeSort) and searching became indispensable, and the systematic study of computational complexity was born [3].

Optimization algorithms also emerged during this period. Early iterative methods—exemplified by the work of Rosenbrock [24] and Rastrigin [22]—demonstrated that algorithmic approaches could effectively tackle nontrivial mathematical problems. In addition, the rise of probabilistic and randomized algorithms greatly enhanced our ability to manage uncertainty and process large-scale data [33].

2.3 Emergence of Adaptive Algorithms

In recent decades, the development of adaptive algorithms has transformed the landscape of computational methods. Unlike static procedures, adaptive algorithms can modify their behavior in response to feedback from input data or changing environmental conditions. Such self-adjusting techniques are at the heart of many modern machine learning and artificial intelligence systems [6, 2]. Moreover, advances in parallel and distributed computing have enabled these adaptive methods to scale to the challenges posed by big data and real-time processing [32].

2.4 Legacy and Future Directions

The historical evolution of algorithms—from the deterministic procedures of Euclid and Al-Khwarizmi to the adaptive, self-improving methods of today—exemplifies the relentless pursuit of computational efficiency and problem-solving prowess. Looking ahead, emerging fields such as quantum computing [27] and biologically inspired methods promise to further revolutionize algorithmic design. As classical insights merge with modern adaptive strategies, the future of algorithms holds tremendous potential for even greater levels of performance and intelligence.

In summary, the journey of algorithmic development is a testament to human ingenuity and the enduring quest to optimize our methods for understanding and interacting with the world. This legacy not only informs current practice but also inspires the next generation of breakthroughs in artificial intelligence and beyond.

3 History of Machine Learning

Machine learning (ML) has undergone a dramatic evolution over the past several decades, transitioning from simple pattern recognition systems to the sophisticated deep learning architectures that underpin modern artificial intelligence (AI). This historical journey not only reflects an increasing capacity to learn from data but also sets the stage for recent breakthroughs—such as our recursive generative framework for automating AI research.

3.1 Early Machine Learning

The origins of machine learning can be traced back to the mid-20th century. One of the earliest influential models was the *perceptron*, introduced by Rosenblatt [23], which demonstrated how simple adaptive systems could mimic aspects of human learning. In parallel, the formulation of statistical learning theory—pioneered by Vapnik and colleagues [31]—provided a rigorous mathematical framework for understanding generalization and the limits of empirical risk minimization. Early decision tree methods, such as ID3 and C4.5 [21], further illustrated how algorithms could automatically discover interpretable rules for classification and regression tasks.

3.2 The Deep Learning Revolution

The resurgence of neural networks in the 1980s and 1990s was marked by the rediscovery and popularization of the backpropagation algorithm [25]. Although initial networks were relatively shallow, these advances laid the groundwork for deeper architectures. Convolutional neural networks (CNNs), advanced by LeCun and collaborators [15], yielded state-of-the-art performance on image-based tasks. The watershed moment came with AlexNet [13], which achieved breakthrough results in the ImageNet challenge, prompting widespread adoption of deep learning. Over time, researchers extended deep networks to various modalities (e.g., audio, text) and demonstrated increasingly sophisticated capabilities [14].

3.3 Modern Advances: GANs, Reinforcement Learning, and Beyond

Beyond supervised learning, *generative adversarial networks* (GANs) [7] revitalized the field of generative modeling by introducing two competing networks (generator and discriminator). Meanwhile, *reinforcement learning* (RL) combined with deep architectures [17] enabled models to master complex tasks exemplified by AlphaGo and AlphaZero [28, 29], surpassing human performance in challenging board games. This era also saw the rise of *foundation models*—large-scale pre-trained networks such as transformers [32], which excel at adapting to varied tasks with minimal fine-tuning.

3.4 Meta-Learning and Automated ML

Another influential trend has been the rise of *meta-learning* [6] and *AutoML* systems [5], which automate tasks like hyperparameter tuning, model selection, and neural architecture search. By iteratively refining their learning strategies based on performance feedback, these methods reduce reliance on painstaking manual experimentation. Notable achievements include few-shot and in-context learning capabilities [2, 18], in which large models generalize to novel tasks from minimal data or instructions.

3.5 Implications for Recursive Generative Algorithms

This historical progression—from simple perceptrons to complex, large-scale models—provides the conceptual scaffolding for our work. Our recursive generative framework capitalizes on deep and meta-learning strategies to yield AI systems that autonomously refine both their architectures and their training protocols. In doing so, we bridge decades of machine learning advances, from foundational statistical insights [31] to recent breakthroughs in representation and generative modeling [7, 28, 32, 18].

In summary, the trajectory of machine learning has been marked by an expanding capacity to leverage data and self-improve. Our approach continues this legacy by embedding self-optimization within AI research itself—opening new avenues for accelerating and refining the next wave of AI systems.

4 "Hello World" Generative Algorithms

In this section, we provide a practical "Hello World" demonstration of generative algorithms in continuous optimization. We present twenty well-known 2D fitness functions, define each mathematically, and show how to compute a single 0–1 *final comprehensive score* that reflects each function's complexity or ruggedness. Using an LLM to imagine or halucinate new functions is not impressive,

you'll see from the prompt in the appendix driving outputs that are competitive is another matter. To begin this section we will cover in detail many of the famous cost functions that are used for benchmarking new solvers. Keep in mind, these functions, are tied back to a human inventor and many of them had publications because of the novelty and the positive impact to the research communities.

4.1 Classic Optimization Functions

4.1.1 Rosenbrock Function

$$f_{\text{Rosenbrock}} = 100\,(y - x^2)^2 + (x - 1)^2.$$

Domain: $(-5, 10)$. Known for its narrow curved valley ending in the global minimum near $(1, 1)$.

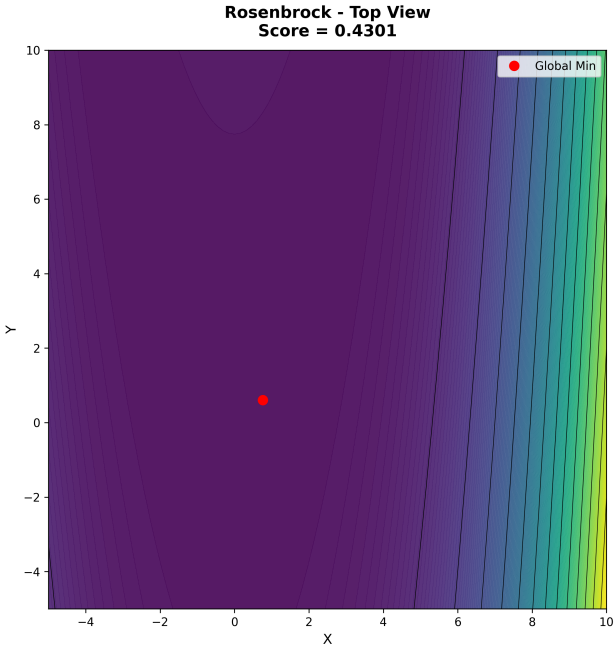


Figure 2: Rosenbrock function.

4.1.2 Rastrigin Function

$$f_{\text{Rastrigin}} = 20 + (x^2 - 10\cos(2\pi x)) + (y^2 - 10\cos(2\pi y)).$$

Domain: $(-5.12, 5.12)$. Displays large numbers of regularly spaced local minima, making it a standard multimodal test.

4.1.3 Ackley Function

$$\begin{aligned} f_{\text{Ackley}} = &-20\exp\left(-0.2\sqrt{0.5(x^2 + y^2)}\right) \\ &- \exp\left(0.5\left[\cos(2\pi x) + \cos(2\pi y)\right]\right) \\ &+ 20 + e. \end{aligned}$$

Domain: $(-32.768, 32.768)$. A widely used benchmark with a nearly flat outer region and a central basin near the origin.

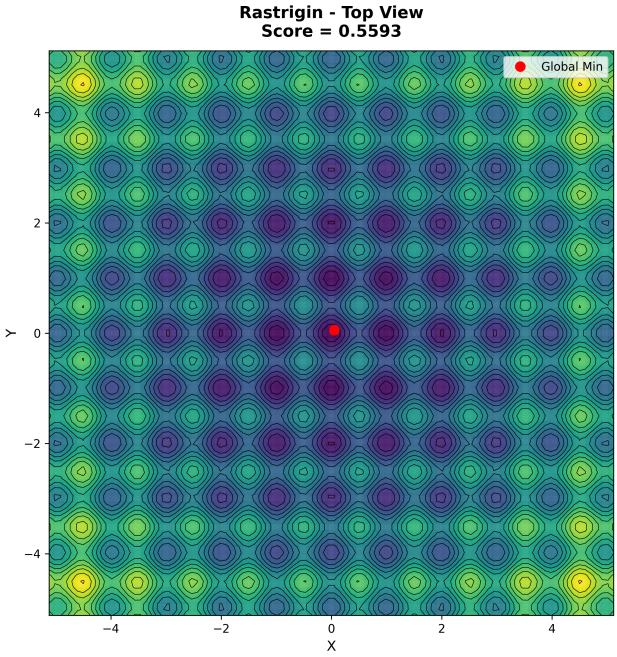


Figure 3: Rastrigin function.

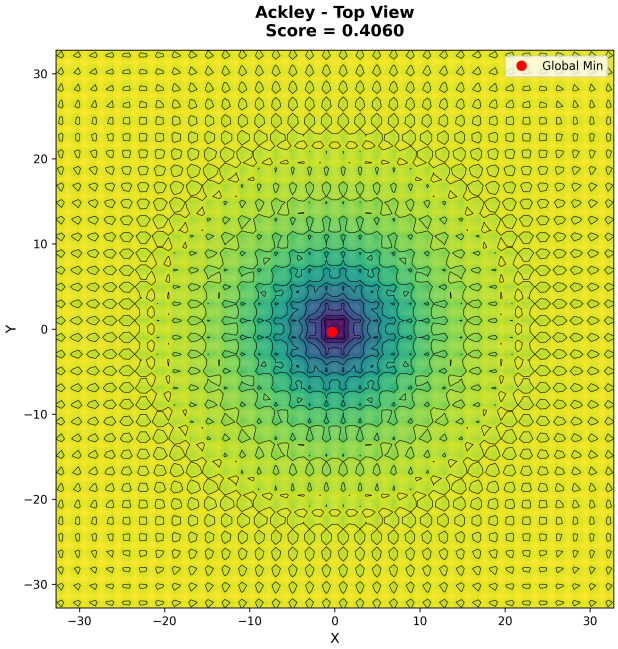


Figure 4: Ackley function.

4.1.4 Sphere Function

$$f_{\text{Sphere}} = x^2 + y^2.$$

Domain: $(-5.12, 5.12)$. A simple convex function with a single global minimum at $(0, 0)$.

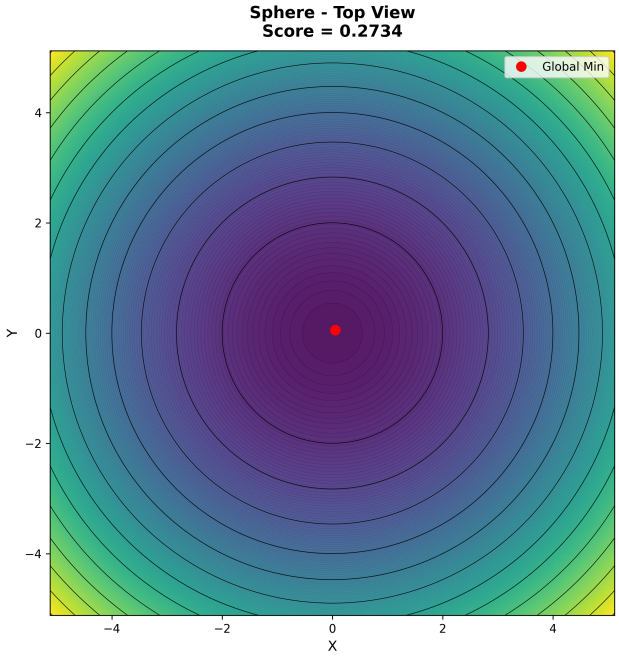


Figure 5: Sphere function.

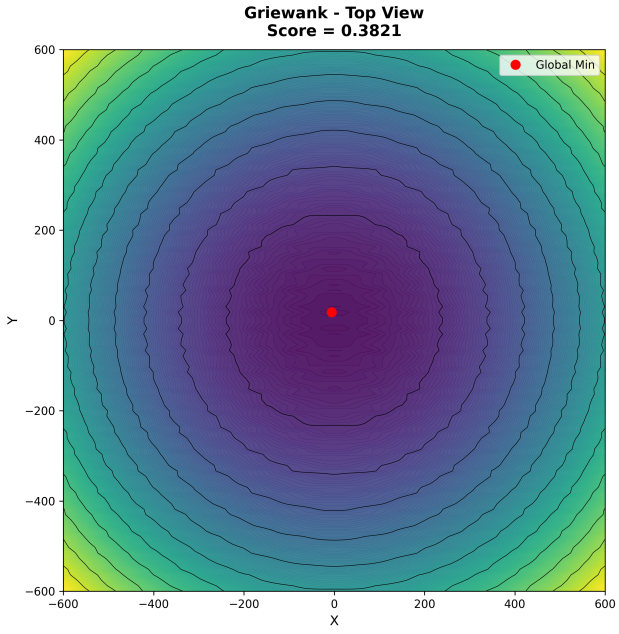


Figure 6: Griewank function.

4.1.5 Griewank Function

$$f_{\text{Griewank}} = 1 + \frac{x^2 + y^2}{4000} - \cos\left(\frac{x}{\sqrt{1}}\right)\cos\left(\frac{y}{\sqrt{2}}\right).$$

Domain: $(-600, 600)$. A scaled multimodal benchmark reminiscent of Rastrigin, but with subtler oscillations.

4.1.6 Schwefel Function

$$f_{\text{Schwefel}} = 418.9829 \times 2 - x\,\sin\left(\sqrt{|x|}\right) - y\,\sin\left(\sqrt{|y|}\right).$$

Domain: $(-500, 500)$. Features a global optimum far from the origin, with many sub-basins along the way.

4.1.7 Lévy Function

$$w_1 = 1 + \frac{x-1}{4}, \quad w_2 = 1 + \frac{y-1}{4}, \tag{1}$$

$$f_{\text{Levy}} = \sin^2(\pi w_1) + (w_1 - 1)^2\left(1 + 10\sin^2(\pi w_2)\right) \tag{2}$$

$$+ (w_2 - 1)^2. \tag{3}$$

Domain: $(-10, 10)$.

A well-known multimodal function with a global minimum at $(1, 1)$.

4.1.8 Alpine Function

$$f_{\text{Alpine}} = |x\sin(x) + 0.1x| + |y\sin(y) + 0.1y|.$$

Domain: $(-10, 10)$. Non-smooth "absolute-value" peaks combined with sinusoidal oscillations.

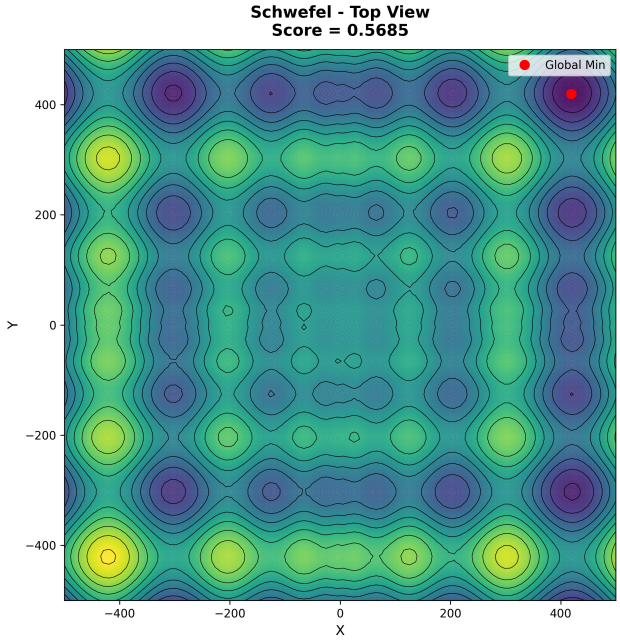


Figure 7: Schwefel function.

4.1.9 Dixon-Price Function

$$f_{\text{Dixon-Price}} = (x-1)^2 + 2\left(2y^2 - x\right)^2.$$

Domain: $(-10, 10)$. A polynomial-based benchmark linking x and y in nested terms.

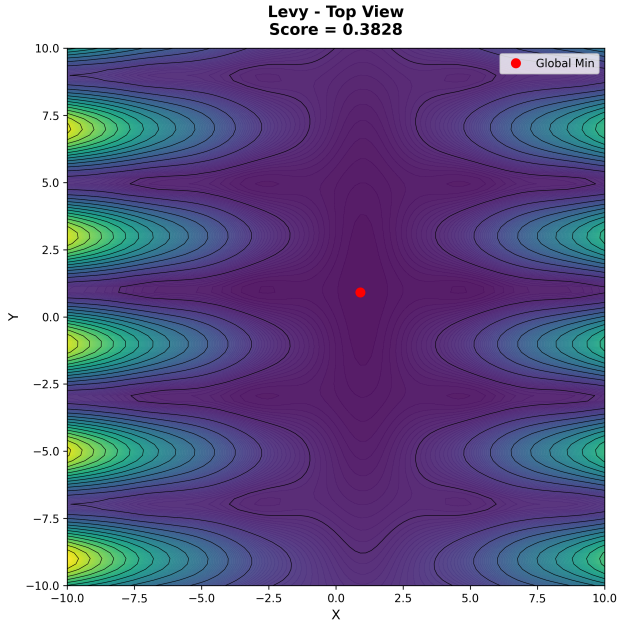


Figure 8: Lévy function.

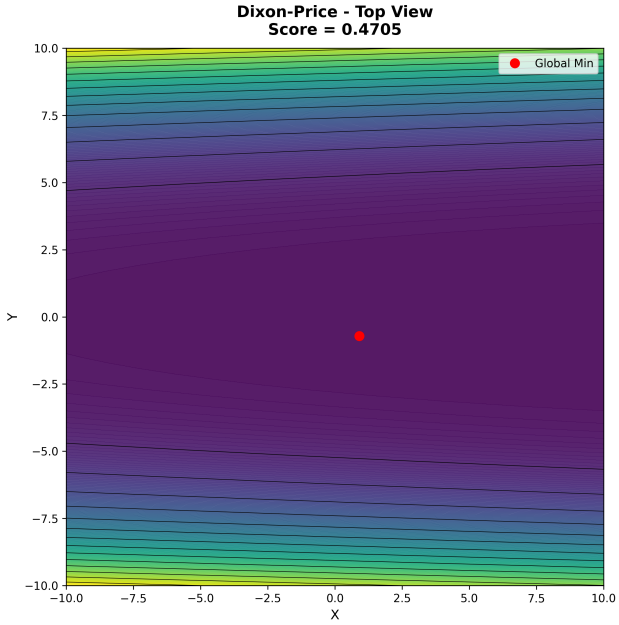


Figure 10: Dixon-Price function.

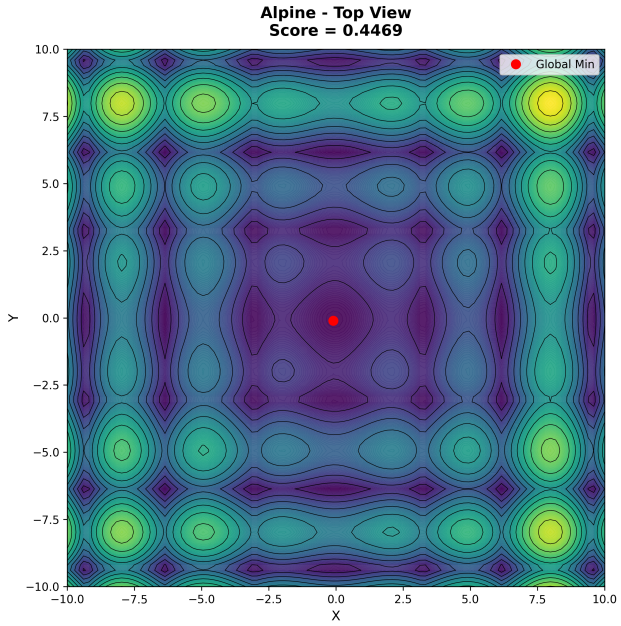


Figure 9: Alpine function.

4.1.10 Michalewicz Function

$$f_{\text{Michalewicz}} = -\left[\sin(x)\sin\left(\frac{x^2}{\pi}\right)^{2m} + \sin(y)\sin\left(\frac{2y^2}{\pi}\right)^{2m}\right], \text{ with } m = 10. \tag{4}$$

Domain: $(0, \pi)$. A spiky multimodal function requiring strong exploration abilities. Two terms that I like are 'intensification', an algorithms ability to chase a gradient or direction and 'diversification', an algorithms ability to avoid local minimum or search for greener pastures.

4.1.11 Bohachevsky Function

Bohachevsky's function is a classical test function defined by:

$$f(x, y) = x^2 + 2y^2 - 0.3\cos(3\pi x) - 0.4\cos(4\pi y) + 0.7\,.$$

It has a global minimum at $(0, 0)$ with $f(0, 0) = 0.7$.

4.1.12 Bent Cigar Function

The Bent Cigar function is often used in high-dimensional optimization studies. In 2D, it is:

$$f(x, y) = x^2 + 10^6\, y^2.$$

Its global minimum is at $(0, 0)$ with $f(0, 0) = 0$.

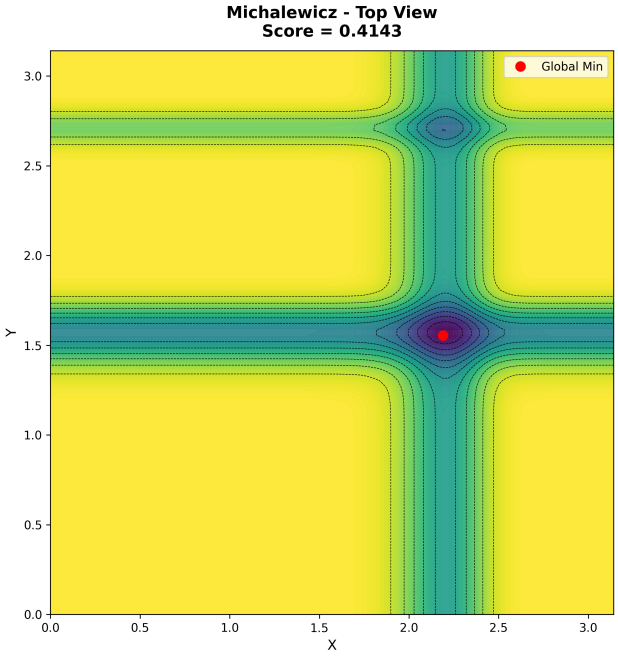


Figure 11: Michalewicz function.

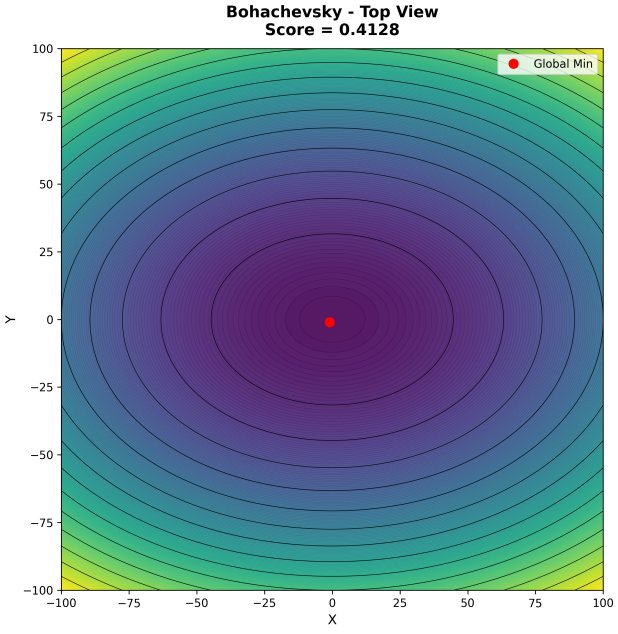


Figure 12: Bohachevsky function.

4.2 Biomechanical Fitness Functions

4.2.1 Hill-Muscle Function

$$f_{\text{Hill-Muscle}} = \frac{(1-x)}{(1+0.25x)} \times \frac{(1-y)}{(1+0.25y)} + 2.0(x^2+y^2).$$

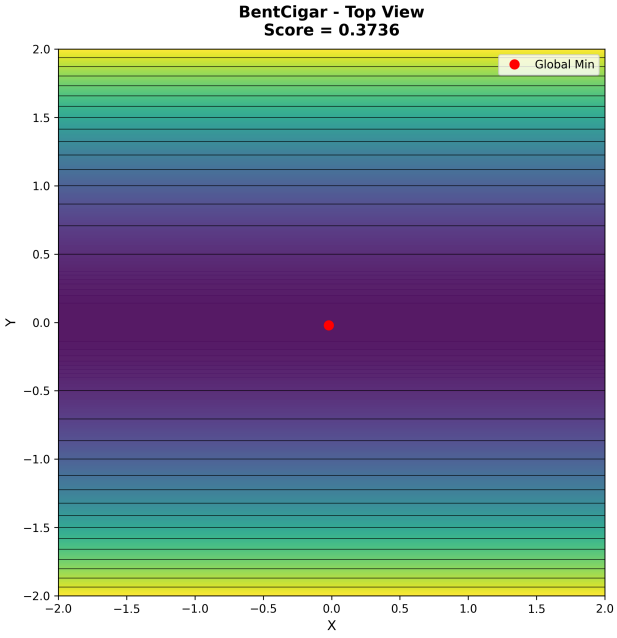


Figure 13: Bentcigar function.

Domain: $(-2, 2)$. Based on A.V. Hill's muscle force-velocity relationship plus a quadratic penalty.

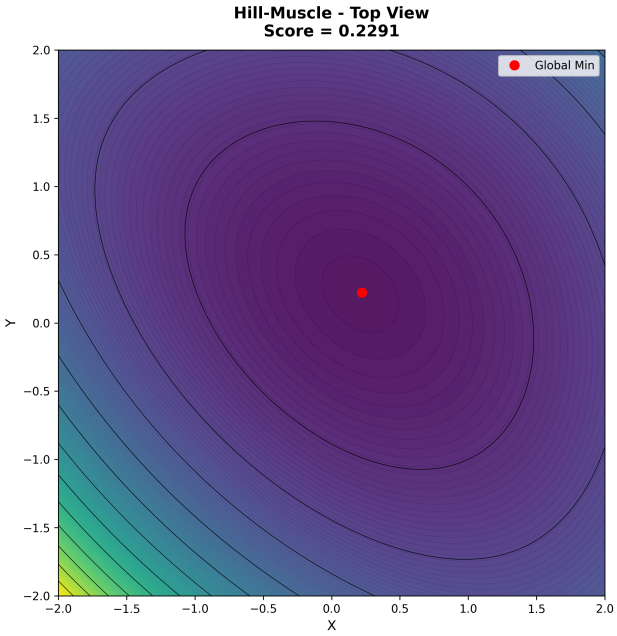


Figure 14: Hill-Muscle function.

4.2.2 Hatze Function

$$f_{\text{Hatze}} = (1-e^{-6x})(1-e^{-6y}) + 0.1(x^2+y^2).$$

Domain: $(-3, 3)$. Derived from muscle activation models with exponential saturation.

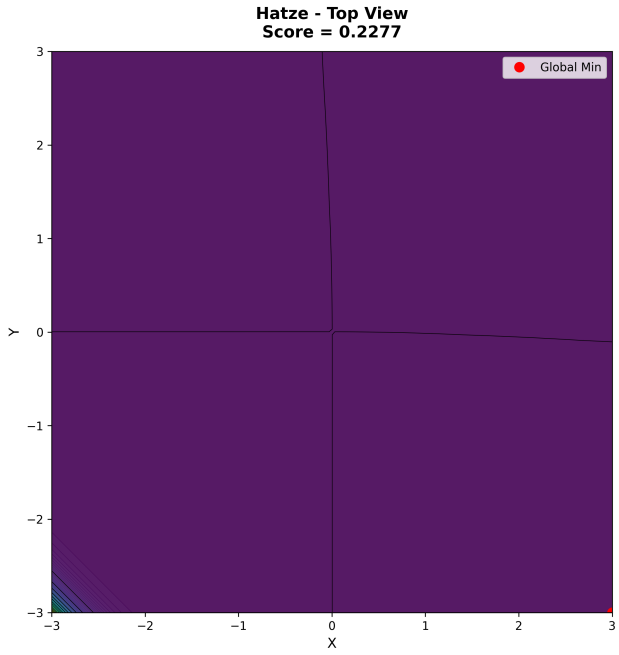


Figure 15: Hatze function.

4.2.3 Huxley Function

$$f_{\text{Huxley}} = 0.5x(1 - y) - 0.1y + 0.2(x^2 + y^2).$$

Domain: $(-4, 4)$. Cross-bridge theory representation with linear and quadratic terms.

4.2.4 Freund Function

$$f_{\text{Freund}} = 2.0\left(e^{0.5x} + e^{0.5y}\right) + 0.1(x^2 + y^2).$$

Domain: $(-5, 5)$. Exponential growth components plus a mild quadratic penalty.

4.2.5 Winter Function

$$f_{\text{Winter}} = (x^2 + y^2)^{\frac{1}{3}} + 0.2\left[\sin(3x) + \sin(3y)\right].$$

Domain: $(-4, 4)$. Combines a fractional power law with sinusoidal modulations, used in biomechanical power analyses.

4.2.6 Minetti Function

$$f_{\text{Minetti}} = 3.8 + 37.5(x^2 + y^2) + 5.4(|x| + |y|).$$

Domain: $(-3, 3)$. A cost-of-locomotion model with both squared and absolute-value penalties.

4.2.7 Alexander Function

$$f_{\text{Alexander}} = (x^2 + y^2)^{0.67} + 0.3(xy).$$

Domain: $(-4, 4)$. Captures mechanical similarity principles with a sub-quadratic exponent.

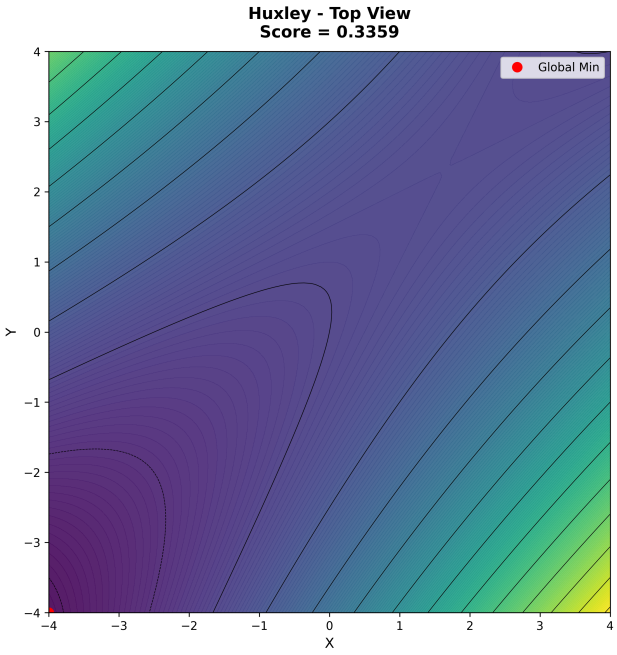


Figure 16: Huxley function.

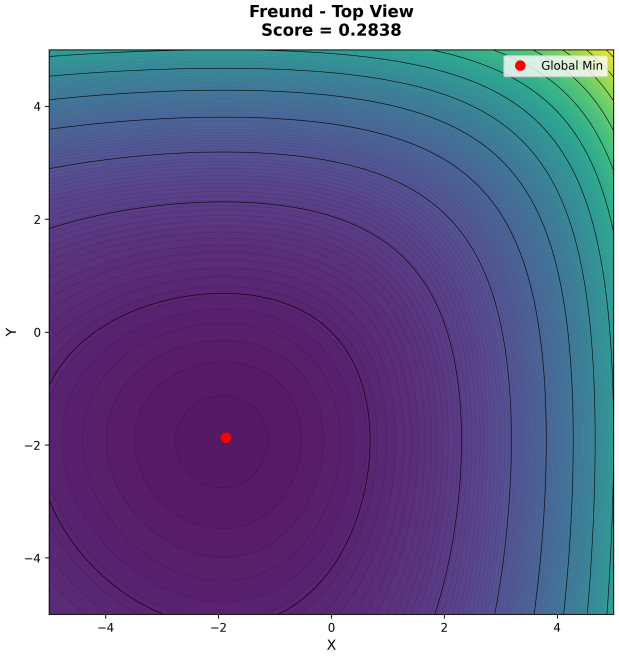


Figure 17: Freund function.

4.2.8 Margaria Function

$$f_{\text{Margaria}} = 9.81\left(\sqrt{x^2 + y^2} + 0.5(x + y)\right).$$

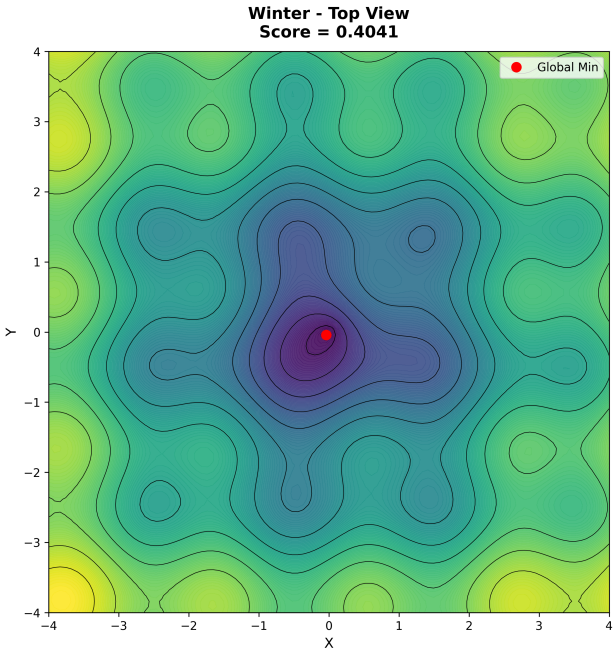


Figure 18: Winter function.

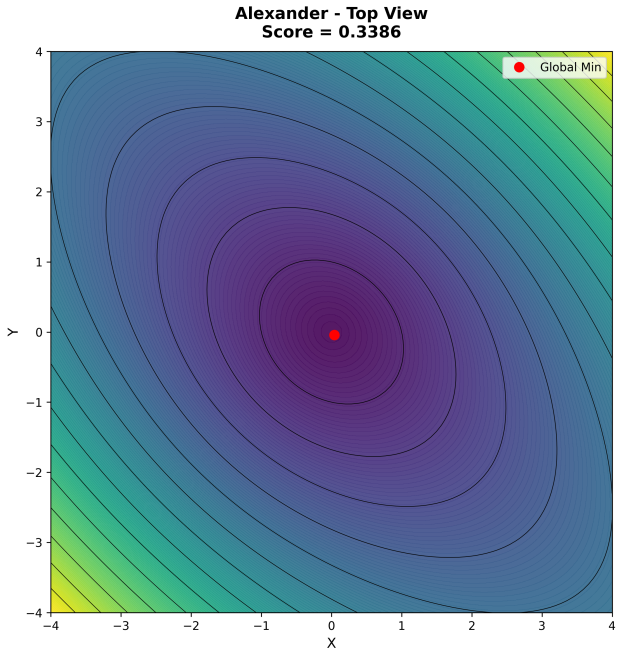


Figure 20: Alexander function.

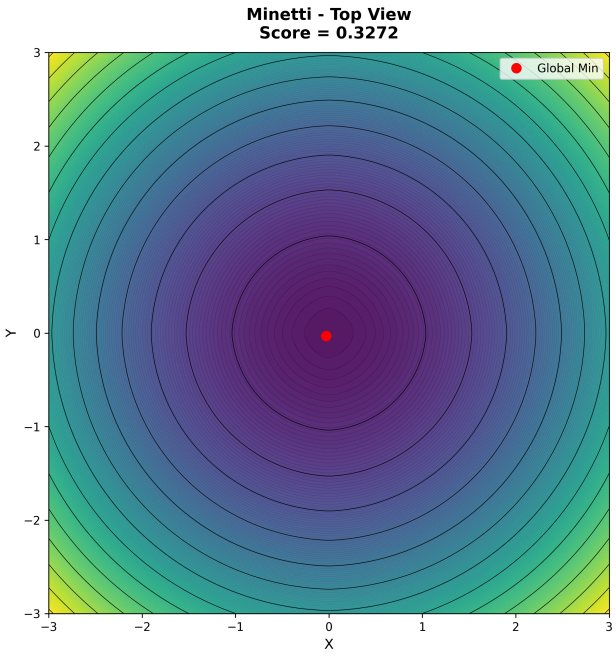


Figure 19: Minetti function.

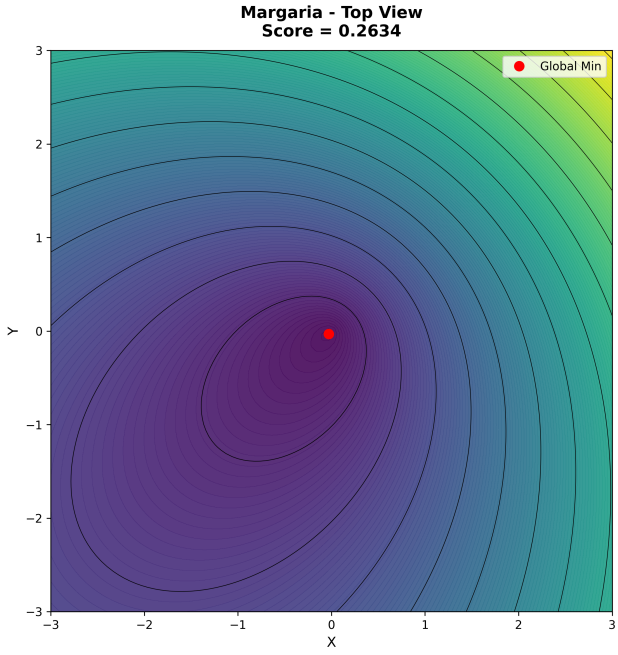


Figure 21: Margaria function.

4.2.9 Wilkie Function

Domain: $(-3, 3)$. From the Margaria–Kalamen power test, includes Earth's gravity constant 9.81.

$$f_{\text{Wilkie}} = \frac{0.3}{(x+0.4)} + \frac{0.3}{(y+0.4)} + 0.1(x^2+y^2).$$

Domain: $(-4, 4)$. Muscle force-velocity-power model containing reciprocal and quadratic terms.

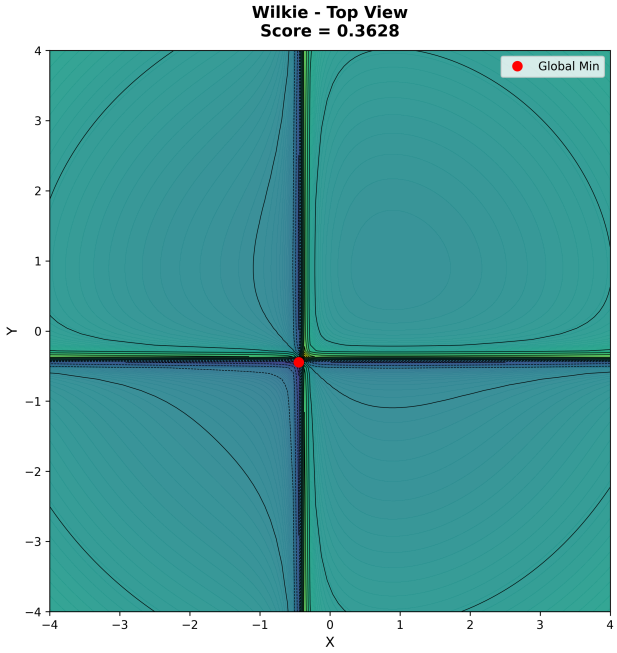


Figure 22: Wilkie function.

4.2.10 McMahon Function

$$f_{\text{McMahon}} = 2.0 \left(\frac{x^2}{3} + \frac{y^2}{3} \right) + 0.2\sqrt{x^2 + y^2}.$$

Domain: $(-5, 5)$. An "elastic similarity" model combining partial squares with a root term.

4.3 Solvers

Now that we have defined a rich body of challenging tests cases we must now find powerful stochastic solvers that are capable of solving these problems. Here are some of the tried and true classics. These algorithms are used in production all over the world, from supply chain at Walmart to hedge fund entry/exit solvers in attempts to maximize Sortino ratios.

4.3.1 Genetic Algorithm (GA)

Genetic Algorithms (GA) are stochastic optimization methods inspired by the principles of natural selection and genetics. They have proven effective in diverse continuous and discrete optimization tasks by evolving candidate solutions over successive generations [16]. A GA typically involves four main operations: selection, crossover, mutation, and replacement. It balances exploitation of good solutions and exploration of the search space, making it robust for many non-convex problems [11].

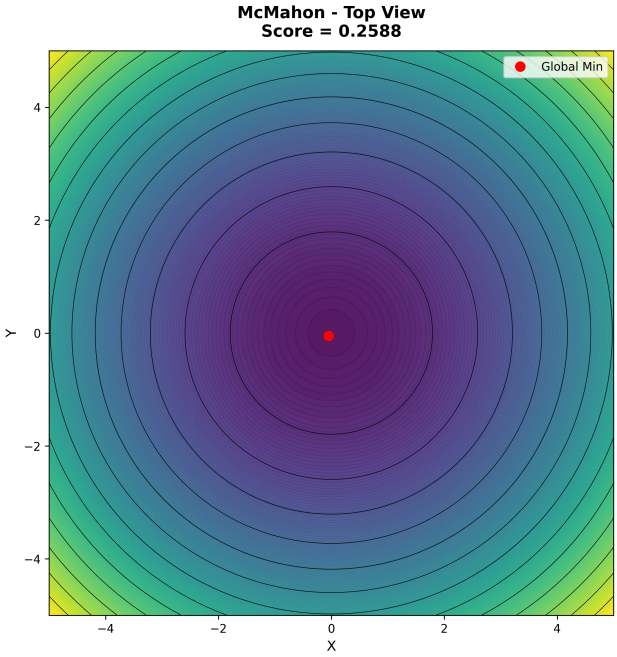


Figure 23: McMahon function.

Algorithm 1 Genetic Algorithm (GA)

Require: Population size $popSize$, crossover probability p_c, mutation probability p_m, function dimension D, max generations G

Initialize population P with $popSize$ solutions in D-dimensional space
for $g \leftarrow 1$ to G **do**
    Evaluate fitness of each individual in P
    Select parents from P based on fitness (e.g., tournament or roulette selection)
    Crossover: with probability p_c, swap partial genes between parents to form offspring
    Mutate: with probability p_m, introduce small random changes in offspring
    Form new population P from best individuals among parents and offspring
end for
return The best solution found (lowest fitness value)

Additional details on GA design and parameter tuning can be found in [16].

4.3.2 Particle Swarm Optimization (PSO)

Particle Swarm Optimization (PSO) is a population-based stochastic optimization technique inspired by social behaviors of bird flocking or fish schooling. Each particle moves through the search space, adjusting its velocity based on its own best-known position and the global best position found by the swarm. This method is known for its simplic-

ity, fast convergence, and relatively few parameters to tune [11].

Algorithm 2 Particle Swarm Optimization (PSO)

Require: Swarm size N, inertia weight ω, personal/-global accel. coeffs. c_1, c_2, max iterations M

Initialize each particle's position $\mathbf{x}_i$ and velocity $\mathbf{v}_i$ randomly
Set personal best $\mathbf{p}_i = \mathbf{x}_i$ and global best $\mathbf{g}$ as the best $\mathbf{p}_i$ among all particles
for $t \leftarrow 1$ to M **do**
  for each particle $i = 1, \ldots, N$ **do**
    Update velocity:

$$\mathbf{v}_i \leftarrow \omega\,\mathbf{v}_i + c_1\,r_1\,(\mathbf{p}_i - \mathbf{x}_i) + c_2\,r_2\,(\mathbf{g} - \mathbf{x}_i)$$

    Update position:

$$\mathbf{x}_i \leftarrow \mathbf{x}_i + \mathbf{v}_i$$

    if $f(\mathbf{x}_i) < f(\mathbf{p}_i)$ **then**
      $\mathbf{p}_i \leftarrow \mathbf{x}_i$ ▷ Update personal best
    end if
  end for
  Update $\mathbf{g}$ to be the best $\mathbf{p}_i$ found by the swarm
end for
return $\mathbf{g}$, the best-known global solution

PSO's success in continuous optimization tasks is often attributed to its balance between exploration and exploitation. For background on general algorithmic strategies, see [11, 3].

4.3.3 Differential Evolution (DE)

Differential Evolution (DE) is a simple yet powerful evolutionary algorithm that relies on mutation based on scaled differences between randomly chosen population members, followed by crossover and selection. It is well-suited for real-valued optimization problems and is known for reliably handling complex, non-linear, and multimodal landscapes.

DE's straightforward parameter set and powerful mutation strategy make it robust for many benchmark and real-world problems. For more on general algorithmic design, see [11, 3].

There is a key reason why optimizers were the first algorithm class where humans were able to demonstrate generative algorithm innovation with LLMs. By reviewing the pseudo code you might see they share similar structure and are very 'hybridizable'. Humans have published papers demonstrating that you can create hybrid algorithms between PSO and GA etc..

4.4 Results and Usage

These cost functions can be used by a variety of solvers to find local or global minima. Well-known stochastic approaches include genetic algorithms [16], particle swarm optimization [11], and differential evolution [11]. Because many of these functions have multiple local minima, advanced methods are often needed for efficient convergence. This plot demonstrates running these listed 'human invented' optimizers across mroe then 40 cost functions and looking at mean rank, meaning compared to an algorithm's peers how often was it the best solution. More robust solvers have lower relative rank scores.

Algorithm 3 Differential Evolution (DE)

Require: Population size $popSize$, scaling factor F, crossover probability p_c, max generations G

Initialize a population P of $popSize$ solutions randomly
for $g \leftarrow 1$ to G **do**
  for each target vector $\mathbf{x}_i$ in P **do**
    Randomly select distinct indices $r_1, r_2, r_3 \neq i$
    Form **mutant vector**:

$$\mathbf{v} \leftarrow \mathbf{x}_{r_1} + F\,(\mathbf{x}_{r_2} - \mathbf{x}_{r_3})$$

    Create **trial vector** $\mathbf{u}$ by crossover of $\mathbf{x}_i$ and $\mathbf{v}$:

$$u_j = \begin{cases} v_j & \text{if rand}(j) < p_c \\ x_{i,j} & \text{otherwise} \end{cases}$$

    if $f(\mathbf{u}) < f(\mathbf{x}_i)$ **then**
      $\mathbf{x}_i \leftarrow \mathbf{u}$ ▷ Selection: replace if improved
    end if
  end for
end for
return The best solution found in P

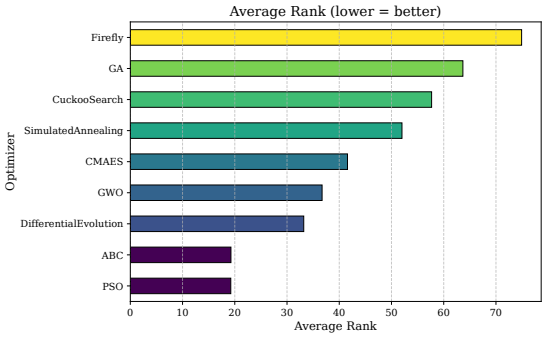


Figure 24: Example ranking of popular optimization solvers based on average performance across selected benchmark functions. Lower rank values indicate better performance.

The relative performance shown in Figure 27 often indicates that certain modern solvers, such as advanced variants of swarm-based methods, can outperform more classical methods for complex, multimodal problems. For example, adaptive butterfly or improved firefly algorithms may converge more reliably than older genetic algorithm

frameworks on certain rugged functions (e.g., the Rastrigin [22] or Michalewicz [16] functions).

4.5 Measuring Complexity and Difficulty

We take a dual approach to discussing difficulty. You can obtain that directly from a solver benchmark, however you can also consider other surface behaviors that are worthy of our attention. The reason we do this, is having a surface where everything is 0 and I have a random pixel that is very small or large number is extremely diffuclt requiring an exhaustive search, but that problem is not interesting. So by combining the theory below with the actual performance, this increases the changes that our final benchmarking scores have real meaning and value. To quantify function difficulty, we define a multi-measure complexity index explain in the following sections.

4.5.1 Log Difficulty Score for Benchmark Functions

In this section, we introduce and detail the derivation of our *Log Difficulty Score* for two-dimensional benchmark functions commonly used in global optimization research (e.g., [22, 1, 24, 26]). Given a function $f(x, y)$ defined over a rectangular domain $\Omega \subset R^2$, we discretize the domain into a grid of resolution $N \times N$, generating the mesh

$$\begin{aligned} X &= \{x_1, x_2, \ldots, x_N\}, \quad x_i \in [a, b], \\ Y &= \{y_1, y_2, \ldots, y_N\}, \quad y_j \in [c, d], \end{aligned} \tag{5}$$

where $\Omega = [a, b] \times [c, d]$. We then form the matrix

$$Z \in R^{N \times N}, \quad Z_{ij} = f\big(x_i,\, y_j\big).$$

Our difficulty metrics rest on analyzing this matrix Z to approximate various complexity-related features of the underlying function. To obtain a single scalar measure of complexity, we aggregate multiple sub-metrics into a composite *complexity score*, which is then mapped to a *log difficulty score* for downstream usage in function selection and benchmark curation. Below, we provide a step-by-step explanation.

4.6 Sub-Metrics in the Composite Complexity Score

Let $Z_{i,j}$ represent the value of $f(x_i, y_j)$ on the grid, and let $\mathcal{M} = \{(i, j) \mid 1 \le i, j \le N\}$ be the set of all mesh points. We define a core set of sub-metrics $\{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_{13}\}$ that quantify different aspects of the function's geometry, ruggedness, and distribution. Each sub-metric is normalized to lie between 0 and a small constant (e.g., $1/13$) so that, when summed, they do not exceed 1.0.

1. **Local extrema count.** Identify local maxima or minima by comparing each $Z_{i,j}$ to its neighbors in a 3×3 window. Let LM be the set of these local extrema. Define a pivot scale p_e (e.g., $p_e = 0.005\, N^2$), then

$$\text{raw1} = 1 - \exp\Big(-\tfrac{|\,\text{LM}\,|}{p_e}\Big), \quad \mathcal{S}_1 = \alpha(\text{raw1}).$$

A larger number of local extrema indicates a more complex landscape.

2. **Gradient ruggedness.** Approximate partial derivatives using finite differences:

$$\frac{\partial Z}{\partial x}, \quad \frac{\partial Z}{\partial y},$$

then let

$$G_{i,j} = \sqrt{\Big(\tfrac{\partial Z}{\partial x}\Big)^2 + \Big(\tfrac{\partial Z}{\partial y}\Big)^2}.$$

Compute the standard deviation σ_G of $G_{i,j}$ over Ω and define a pivot $\gamma > 0$, e.g., $\gamma = 2$:

$$\text{raw2} = 1 - \exp\Big(-\tfrac{\sigma_G}{\gamma}\Big), \quad \mathcal{S}_2 = \alpha(\text{raw2}).$$

3. **Distribution-based entropy.** Interpret Z as a histogram of values with K bins, and let p_k be the fraction in bin k. The (Shannon) entropy is

$$H_{\text{dist}} = -\sum_{k=1}^{K} p_k \ln(p_k + \varepsilon),$$

normalized by $\ln(K)$. We set

$$\mathcal{S}_3 = \alpha\Big(\tfrac{H_{\text{dist}}}{\ln(K)}\Big).$$

4. **Basin measure.** Let $Z_{\min}$ and $Z_{\max}$ be the min and max of Z. Define $t = Z_{\min} + \beta(Z_{\max} - Z_{\min})$ with $\beta \in (0, 1)$, and compute the fraction of grid points below t. We set

$$\mathcal{S}_4 = \alpha\big(1 - \text{FracBelow}\big).$$

5. **Tile-based evenness.** Divide the grid into $r \times r$ tiles, count "interesting" points (e.g., local maxima or large gradient) in each tile, then measure how evenly these points are distributed across tiles. This yields

$$\mathcal{S}_5 = \alpha(\text{raw5}),$$

where raw5 measures the deviation from uniform distribution.

6. **Multi-scale correlation.** Create smoothed versions of Z via Gaussian filters $\sigma_1, \sigma_2, \sigma_3$. Compute correlation $\rho(\sigma)$ with the original Z, then average:

$$\overline{\rho} = \tfrac{1}{3}\sum_{k=1}^{3} \rho(\sigma_k), \quad \text{raw6} = 1 - \overline{\rho}, \quad \mathcal{S}_6 = \alpha(\text{raw6}).$$

7. **Value diversity.** Let U be the number of unique $Z_{i,j}$ values. We define

$$\text{raw7} = \min\Big(1,\ \tfrac{U/N^2}{0.3}\Big), \quad \mathcal{S}_7 = \alpha(\text{raw7}).$$

8. **Expression simplicity.** If f stems from a symbolic expression E with token count $|E|$, define

$$\text{raw8} = \exp\Big(-\tfrac{|E|}{p_s}\Big), \quad \mathcal{S}_8 = \alpha(\text{raw8}).$$

9. **Second-derivative variance.** Approximate the Laplacian $\nabla^2 Z$ by finite differencing. Let σ_{∇^2} be its standard deviation:

$$\text{raw9} = 1 - \exp\Big(-\tfrac{\sigma_{\nabla^2}}{\gamma'}\Big), \quad \mathcal{S}_9 = \alpha(\text{raw9}).$$

10. **Fourier-based frequency richness.** Perform a 2D DFT on Z, interpret magnitudes as probabilities, and compute entropy H_{freq}. Then

$$\mathcal{S}_{10} = \alpha\Big(\tfrac{H_{\text{freq}}}{\ln(N^2)}\Big).$$

11. **Fractal-like boundary measure.** Threshold Z at median and 75th percentile to measure boundary complexity. Let BC_1 and BC_2 be boundary counts; define

$$\text{bc\_ratio} = \tfrac{\text{BC}_2}{\text{BC}_1 + \varepsilon}, \tag{6}$$

$$\text{raw11} = 1 - \exp\big(-|\,\text{bc\_ratio} - 1|\big), \tag{7}$$

$$\mathcal{S}_{11} = \alpha(\text{raw11}). \tag{8}$$

12. **Cross-derivative variance.** Let $D_\times = \dfrac{\partial}{\partial y}\big(\dfrac{\partial Z}{\partial x}\big)$. Its standard deviation $\sigma_{D_\times}$ defines

$$\text{raw12} = 1 - \exp\Big(-\tfrac{\sigma_{D_\times}}{\gamma''}\Big), \quad \mathcal{S}_{12} = \alpha(\text{raw12}).$$

13. **Peak kurtosis alignment.** Let K_Z be the sample kurtosis of all $Z_{i,j}$, and let $K^* = 1.5$:

$$\text{raw13} = \exp\big(-|K_Z - K^*|\big), \quad \mathcal{S}_{13} = \alpha(\text{raw13}).$$

4.7 Composite Score and Log Transformation

Summing these sub-metrics yields the aggregated *complexity score*:

$$S_{\text{complex}}(Z) \;=\; \sum_{i=1}^{13} \mathcal{S}_i \;\in\; [0, 1]. \tag{9}$$

In rare cases, extremely large or small values of $Z_{i,j}$ can skew these metrics, so a small penalty factor can be introduced if $E[|Z_{i,j}|]$ exceeds a threshold (e.g., 1000). Finally, we define the **Log Difficulty Score**:

$$D_{\log}(f) \;=\; \ln\big(S_{\text{complex}}(Z) + 10^{-12}\big), \tag{10}$$

which places greater emphasis on differences near zero. Higher $D_{\log}(f)$ indicates a more elaborate and challenging landscape.

4.8 Usage in Adaptive Systems and Benchmark Ranking

In evolutionary approaches or genetic programming coupled with agents, $D_{\log}(f)$ helps identify which newly evolved functions present real difficulty for standard solvers. By focusing on functions with higher $D_{\log}$ scores and lower success rates, one can curate a next-generation set of benchmarks. This ensures that new functions indeed push the boundaries of optimization algorithms, a core goal in global optimization research [8].

5 Extended Results on Agent-Generated Functions

It is challenging to fully convey the complexity behind the following prompt (moved to appendix). If you only want to produce random $f(x, y)$ strings from a language model, that alone is unlikely to yield truly intricate functions. Even if you attempted a brute-force approach with a vast number of random prompts, you would rarely produce the kind of sophisticated functions generated by the carefully refined prompt below. These functions achieve very competitive scores on our log difficulty metric. Substantial effort and iterative testing were required to perfect this prompt (see appendix).

We then apply this prompt repeatedly. After many trials, we occasionally encounter functions like the following:

5.0.1 a152b7 Function

Let us define the following terms:

$$\begin{aligned}
T_1 &= e^{\cos(10\pi x)+\cos(10\pi y)} \\
T_2 &= \sin^2(x^2+y^2)\cdot\sqrt{x^2+y^2+10^{-8}} \\
T_3 &= (x^2+y^2)^{1/3} \\
T_4 &= \log(|xy|+10^{-8}) \\
T_5 &= \frac{x^4+y^4}{10} \\
T_6 &= \sin(10(x^2+y))\cos(5\sqrt{x^2+y^2}) \\
T_7 &= 10^6(x^2+2y^2)
\end{aligned}$$

Then the function f_{a152b7} is defined as:

$$f_{\text{a152b7}} = T_1\cdot(T_2+T_3+T_4) + T_5 + T_6 + T_7$$

Domain: $((-1, 1)^2)$. A complex multimodal function with log difficulty score of 8.04. Combines oscillatory behavior with steep parabolic scaling.

We could share additional examples from this process, but most prompt-generated functions are not particularly interesting. As shown in Figure 27, functions with especially high log difficulty scores are rare. On average, one would need to make hundreds of thousands of calls to "get lucky" and obtain a function with a genuinely intricate landscape. At that rate, it might seem that language models are not especially powerful for discovering new, hard-to-optimize functions.

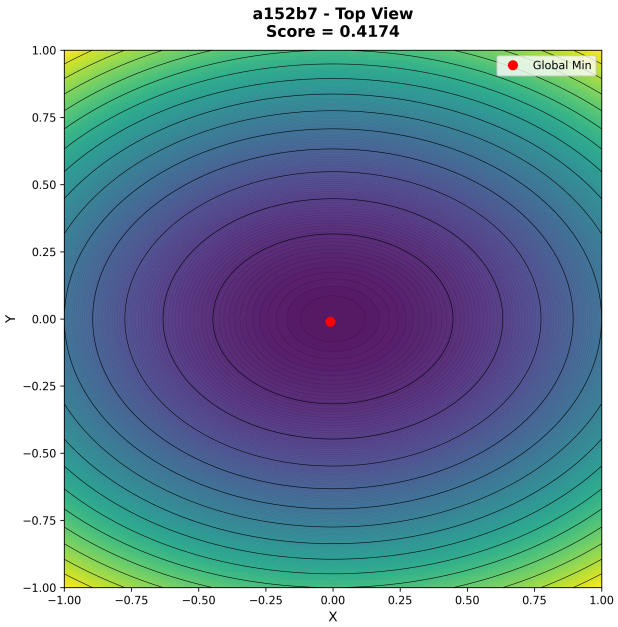


Figure 25: a152b7 function.

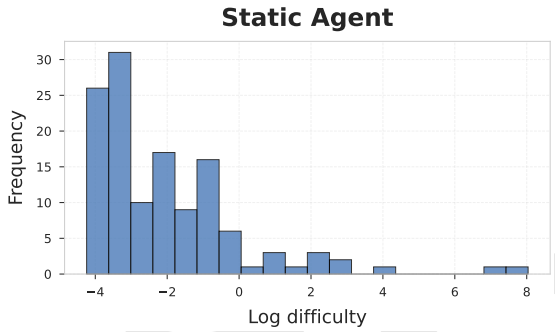


Figure 26: Scoring distribution from repeated LLM calls using the same prompt. Most generated functions yield lower difficulty scores, with only a few achieving very high complexity. This behavior feels like weak autonomous innovation, casual fishing for luck rather than actively chasing a goal.

6 Adaptive Agents

In the previous experiments, we employed static agents, which did not benefit from any evolutionary mechanism to propel better solutions forward. By contrast, if we combine the notion of agent-based simulation with *evolutionary systems*—specifically, genetic programming [12]—we can introduce populations of solutions that adapt and improve across generations. This often leads to what feels like "runaway innovation," as each iteration leverages prior successes to refine future guesses.

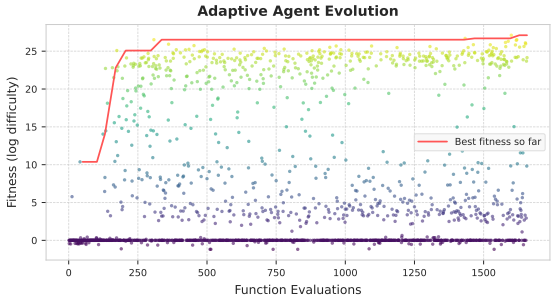


Figure 27: Evolutionary drivers chase higher fitness, allowing adaptive agents to outperform static ones.

By examining the results, we see a dramatic departure from the performance plateau often observed with static agents. Our adaptive agents exploit historical information and use evolutionary pressure to guide exploration. This demonstrates that using large language models (LLMs) to invent simple yet innovative algorithms is indeed feasible. Furthermore, we are confident this approach can extend beyond generating mere cost functions: the same evolutionary framework can be applied to automating the design of optimizers themselves, unlocking even more powerful discoveries.

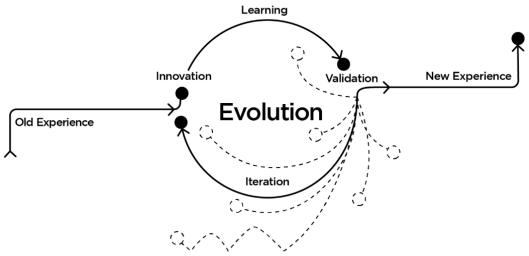


Figure 28: The power of evolutionary cycles benefits from LLM hallucinates, every hallucination is a potential innovation

As humans move away from writing prompts (we were never qualified anyway) they will find themselves spending more time on the goal phase. With these types of autonomous innovaiton cycles having the right goals and metrics are the difference between success and failure. The other key thing with evoluationary cycles is the compounding, when meaningful traction is found, that experience is applied to future exploration. This is where fishing becomes chasing, chasing a goal faster and faster. Hopefully, you can see from this section, if your dream in life was to invent the next best surface for optimizers to test you have some serious competition. With the log difficulty numbers coming off of the adaptive system I would argue it is pretty evident a human can no longer be competitive against this fitness criteria. A key thing to consider is the cost of a bad idea, an AI researcher might spend weeks, months or in the case of a PhD years chasing a bad ideas. With these

approachs millions ideas can be tested and the cost of a bad idea is free falling to zero.

7 Generative Optimizers

If you thought inventing new competitive surfaces with higher difficult scores was impressive just wait. Now that we have demonstrated that generative algorithms can be applied to invent new functions, we will attempt something far more ambitious: *automatically designing optimizers themselves*. Using adaptive agent-based generative approaches, we have produced a collection of *novel global optimizers*, none of which were handcrafted by human researchers. Remarkably, many of these automatically-generated optimizers *outperform* classical, human-invented methods by a significant margin. A critical point when you review these performance results is: if you can find one you can now find a million. The human bottleneck has been removed for discovering new math, new algorithm structure with the primary limiter being compute which is also dropping in price 90% year over year.

7.1 Experimental Results

Table 1 shows the average rank of each optimizer across a battery of test problems (comprising classical and newly generated benchmark functions). Lower numbers indicate better performance on average. Notable generative optimizers, denoted by hashes, climb to the top of the ranking, outstripping classic methods such as Particle Swarm Optimization (PSO), Artificial Bee Colony (ABC), Differential Evolution, and even more sophisticated methods like CMA-ES.

Table 1: Average rank per optimizer. Lower rank is better. Optimizers labeled by short hash (e.g., `afaaac53`) are automatically generated by our agent-based approach. Classical methods are in plain text. Um.. Houston we have a problem..

Optimizer	Avg. Rank
afaaac53	9.556757
65481bcd	29.513514
15fdb014	39.610811
6bf00a18	43.427027
621b5cf3	45.881081
93f9422b	52.800000
PSO	54.654054
125ec7c9	59.259459
955551cb	60.983784
ABC	61.664865
DifferentialEvolution	87.075676
GWO	99.870270
CMAES	113.886486
SimulatedAnnealing	139.421622
CuckooSearch	147.021622
GA	160.113514
Firefly	175.124324

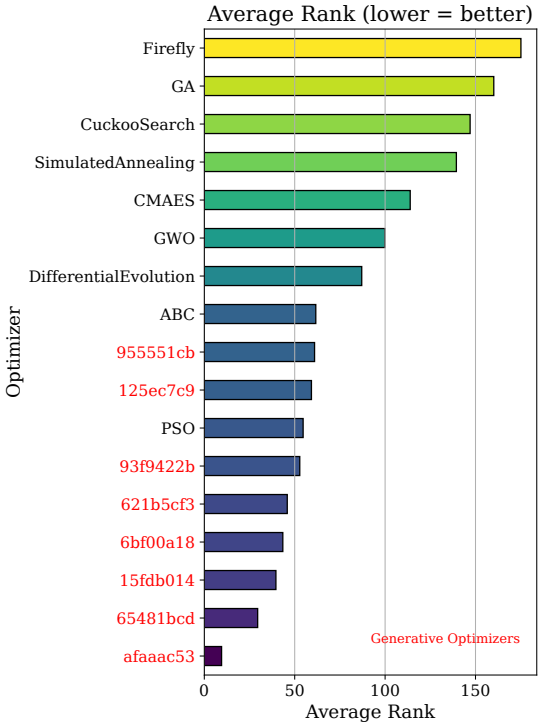


Figure 29: A bar chart visualization of average optimizer rank. Several agent-generated methods (shown as hashed IDs) consistently outperform human-invented optimizers.

These results highlight the *profound* implications of generative design in the domain of optimization. Where classical optimizers might exploit well-understood heuristics, our agent-based generators freely explore a *vast search space of possible algorithms*, synthesizing and testing novel approaches at scale. As evidenced by the ranks, many emergent designs surpass widely used standard optimizers. In principle, this approach to automated optimizer design can be extended indefinitely, yielding an open-ended discovery process for high-performance search methods.

7.2 Representative Optimizer: `6bf00a18`

Below is an illustrative pseudo-code listing for one of our super-human generative optimizers, identified by the hash `6bf00a18`. This optimizer integrates ideas from differential evolution, stochastic tunneling, and dynamic parameter adjustment in a *hybrid quantum-classical* style (see appendix). For someone who has been an AI researcher for most of my career, this type of work would have taken months if not years. I would argue now it is beyond years, humans have saturate our idea space and no human is going to deliver the next big breakthrough in optimization.

7.3 Implications and Future Directions

Generative optimizers represent a leap beyond human-crafted approaches. The possibility of producing indef-

initely many novel optimization algorithms opens the door to *algorithmic creativity at scale*. By leveraging adaptive agent-based strategies and meta-learning, future systems can autonomously explore the space of *all possible* search procedures, inventing specialized optimizers for any domain. The real hunt now is pushing for new ideas, and we're actively doing research to escape the bias that is trained into LLMs. This method thus holds profound implications across science and engineering, where efficient global search remains a core challenge. A very important point here, is this demonstrates innovation without the human experts, without the mathemtician. If you can find one optimizer you can find millions, your only limit is now compute. If we can tackle cost functions, and optimizers what else can we improve? Also, it is fun to point out that you now have a wonderful dance between the generative fitness functions and the generative algorithms where both of them can work to drive and improve innovation on both fronts. For any current problems requiring optimization, limiting that to 'human invented' algorithms is no short sighted when additional gains were possible.

8 Generative Machine Learning

Most of you reading this paper don't care about fitness functions or optimizers, you are here for the steak. The steak is generative machine learning. If we can invent new optimizers using these techniques can we blow open the door of innovation in the AI space by disrupting the stale AutoML techniques of AutoGluon, Catboost, LightBGM, etc.. The think that success in the world optimizers would carry over to machine learning is naive. Machine learning methods are NOT 'hybridizable' like optimization, this means a lot more work and dependence on adaptive agents and their internal knowledge structures to break through the walls that surround the ML community today. The most recent announcement of TabPFN is honestly one of the only interesting things that has happened in the AutoML space recently.

In the future of machine learning you have two paths, left forcing everything into large LLM type models, or right chasing new math, new algorithms, new structure. Success? Sucess will be right down the middle, and because we are not bottlenecked by humans and brilliant mathematicians we are bottlenecked by data. Early research around generative machine learning failed when large synthetic datasets were used to drive algorithm innovation. A very important thing to call out, generative algorithms are different than AutoML and really machine learning or GenAI. Machine learning trains weights to fit historical data, generative machine learning modifies code in realtime to maximize a goal, similar to what we did before with the optimizers.

8.1 Generative Machine Learning FAQ

There is a lot of confusion around this so it doesn't help to spell it out for the reader.

- **Generative Machine Learning is NOT AutoML v2)**: AutoML allowed a non-expert data scientist to test many models, hyper param tune them, and select the best models. As these systems became more sophisticated they attempted to push their limits with voting, stacking, and weighting systems. Generative machine learning allows us to go to the super market, we are no longer limited by the ML incredients in the local kitchen. The number of popular ML algorithms is a very small count.
- **Generative Machine Learning is NOT GenAI**: GenAI has become a catch all category, using LLMs to build next generation tabular classifiers is one approach. Generative machine learning opens up the door for potentially discovering new math that humans missed. Humans have tried to invent the next transformer, I'm convinced they will not, generative machine learning will.

8.2 Role in Automated AI Research

Generative models play a crucial role in automated AI research by accelerating code and increasing the knowledge availabe to the human researcher. Why stop there? The human researcher should just be an adaptive agent with a clear goal and a fitness that it can chase. This work demonstrates that ombitous algorithms such as optimizers and all of machine-learning are up for grabs to the faster computer. It is important to call out that past attempts have been done to tune neural network designs, but those approaches were constrained. Generative machine learning removes the gates, the fences, there are no rules. Every new algorithm startes from a blank slate and evolves to maximize an outcome.

8.3 Challenges and Future Directions

While generative ML opens new possibilities, several challenges remain:

- *Mode Collapse and Limited Diversity*: GANs and other models sometimes fail to capture the full complexity of data distributions, generating repetitive or overly similar samples.
- *Computational Overhead*: Training large generative models can be expensive, emphasizing the need for efficient meta-learning and resource management.
- *Evaluation Metrics*: Assessing generative quality remains an open question, particularly for tasks without straightforward perceptual or statistical measures.

Despite these hurdles, generative machine learning continues to drive innovation in automated model development. A common question that is assumed with generative machine learning is that an LLM agent is being used to

build best-in-class AutoML pipelines. That would be agent drived AutoML, generative machine learning starts with a blank slate. Something that humans have never had available is the potential to invent algorithms in real-time unique to a customer, to a datasets. Many real-world datasets are unique enough that they will discover better algorithms, unique to them. This also drives that point home, it is all about the data, real-world data drives the future innovation, not the human. For the following benchmarks we limited the AutoML methods to 600s run-time for AutoGluon and AutoTabPFN. A more detailed appendix with all model parameters that were used for testing will be coming soon, but any concerns about sub-par model param setup should be mitigated by the presence of AutoGluon and AutoTabPFN in the following bencmarks.

9 Feature Reduction for Controlled Benchmarking

Benchmarking sophisticated tabular learning algorithms (such as TabPFN, AutoGluon, or generative modeling approaches) against real-world datasets sometimes leads to artificially inflated performance scores. In some cases, classification tasks are trivially separable, resulting in near-perfect AUC metrics. Consequently, such "too-easy" scenarios fail to exercise the more powerful properties of advanced methods. To counteract this, we implement a controlled feature-reduction strategy, systematically removing the *most discriminative* features until the out-of-sample AUC drops below a chosen threshold (e.g., 0.80). By doing so, we create more challenging real-world benchmarks that better reflect the limits and comparative advantages of emerging algorithms. Also, since our goal with this technology is to drive real-impact we had little interest in testing on Kaggle style datasets which often are not a good depiction of the real-world. For this test we curated real-world datasets and remove the top features to get the starting validation scores under the 0.8 AUC CatBoost threshold before starting the benchmark.

9.1 Motivation

When datasets inherently yield very high AUC scores (close to 1.0), comparing multiple algorithms becomes less informative. These saturated performance levels do not reveal differences in modeling capabilities because all methods converge to similarly perfect outcomes. In contrast, by removing the strongest predictive features, we obtain a variant of the same dataset with lower but still realistic predictive performance. This pushes methods *out of the trivial region* and highlights how well they adapt to limited signal or latent complexities in the data. Such artificially "degraded" benchmarks are especially important when studying:

- **Meta-learning or zero-shot methods**: e.g., TabPFN, which leverages Bayesian ideas and neural networks to directly predict distributions of model parameters.
- **Automated machine learning (AutoML) systems**: e.g., AutoGluon, which adaptively searches for the best pipeline or hyperparameters.
- **Generative algorithms**: e.g., generative adversarial networks (GANs) [7] or diffusion models [9], which may produce synthetic data for augmentation or domain adaptation. Harder classification tasks serve as a better stress test for augmentation or data generation strategies.

9.2 Implementation Details

The `FeatureReducer` class uses:

- **CatBoost** [20] for feature importance estimation due to its powerful handling of categorical variables and robust performance across diverse datasets.
- **Scikit-learn** [19] components for data preprocessing, including `LabelEncoder`, `StandardScaler`, and `train_test_split`.
- **Caching and locking mechanisms** to ensure thread safety (`fcntl.flock`) so multiple parallel experiments can safely reuse intermediate results, minimizing repetitive computations.
- **Incremental batch removal of features**: we remove the highest-importance features in batches, recalculating out-of-sample AUC after each removal. If the AUC drops below a target threshold, the process halts. Also, ALL datasets are reduced to the 10,000 row limit imposed by TabPFN to ensure fair comparison between current capabilities.

Detailed Steps:

1. **Feature Processing**: Each column is inspected to determine if it is numeric or categorical. Columns with very high missing-value ratios (over 10%) are discarded. Low-cardinality categorical features undergo label encoding; high-cardinality features are converted to numeric with potential forced casting.
2. **Scaling and Baseline AUC**: Remaining columns are scaled with `StandardScaler` to normalize distribution. An initial CatBoost model is fitted, and an out-of-sample AUC is evaluated with a standard train-test split (controlled by `cv_folds`).
3. **Feature Importances**: Using the trained CatBoost model's `get_feature_importance` method, we obtain an importance ranking for each feature.
4. **Top-Feature Removal**: Instead of dropping the *least* important features, we remove the *most* im-

portant ones in batches, thereby degrading performance. Specifically, if the current feature set has d features, we remove $\max(1, \lfloor d \times \text{batch\_size} \rfloor)$ of the top-ranked features. This process is repeated until the AUC dips below the chosen threshold (e.g., 0.80) or until we reach a specified minimum number of features.

5. **Logging and Caching**: We record the feature-removal order, intermediate AUC values, and final results. If a dataset path is provided, the reduced dataset and all logs are written to disk under a file lock to prevent race conditions in multi-threaded contexts.

9.3 Use in Benchmarking

By applying this procedure, we systematically produce a less informative feature set from *otherwise high-performing datasets*. In doing so, we impose a more challenging classification boundary and create a more precise testbed for:

- *TabPFN*, which attempts to *learn to learn* tabular datasets in a Bayesian-like manner;
- *AutoGluon*, a strong AutoML approach that might overfit trivial tasks but faces more interesting optimization searches when the data is partially degraded;
- *Generative approaches*, which can be tested on their ability to recover meaningful features or data representations when the classification task is not overly simple.

Such pipelines let us observe the interplay between model capacity and reduced data signal. The approach can be extended or combined with various forms of data augmentation and domain shifts for even more challenging conditions.

9.4 Reducer Conclusion and Outlook

The `FeatureReducer` class presented here forms part of a broader experimental pipeline for rigorous algorithmic benchmarking. By making real-world datasets *less trivial*, we ensure that advanced techniques—be they meta-learners, ensemble-based AutoML systems, or generative models—are evaluated in a setting where performance is not saturated at near-perfect AUC. This methodology provides more fine-grained insights, allowing us to discriminate which approaches genuinely exhibit superior generalization and robustness in non-trivial classification scenarios.

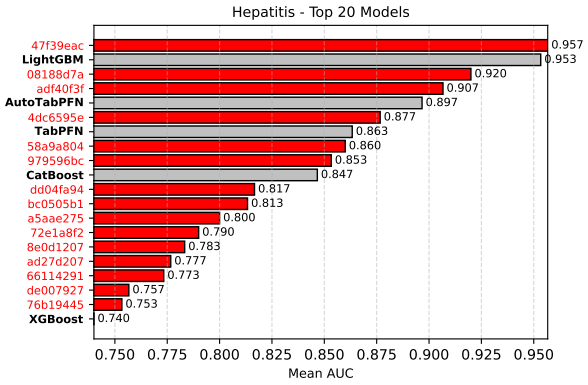


Figure 30: Hepatitis

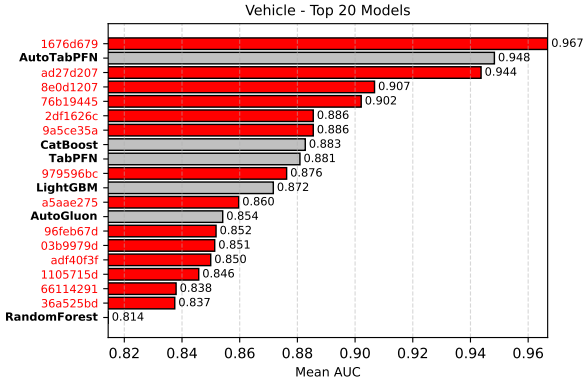


Figure 31: Vehicle

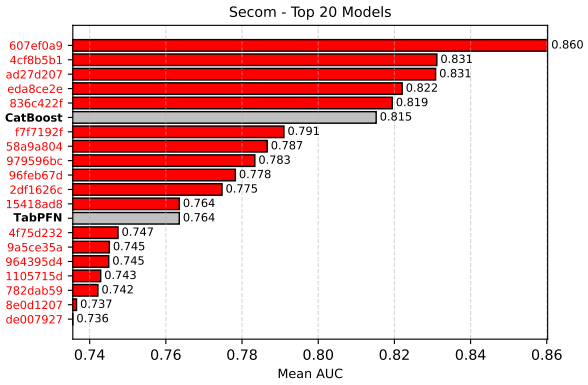


Figure 32: Secom

10 Real-world 10k row Dataset Performance

11 Conclusion

In this paper, we have introduced a novel framework for *automating* AI research by integrating recursive thinking with generative algorithms. Our experiments demonstrate

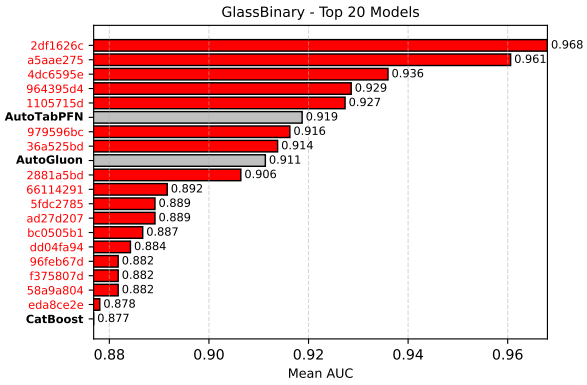


Figure 33: Glassbinary

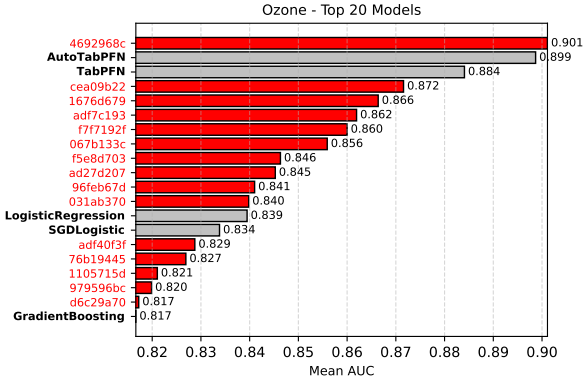


Figure 34: Ozone

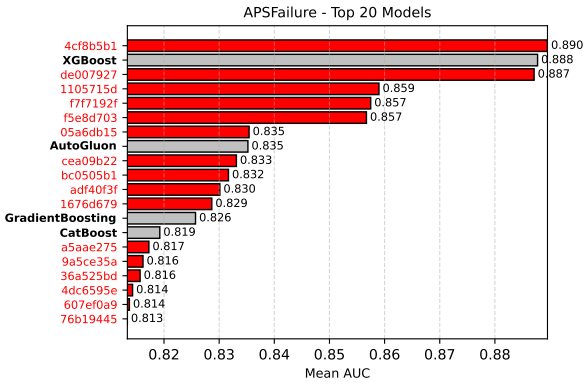


Figure 35: Apsfailure

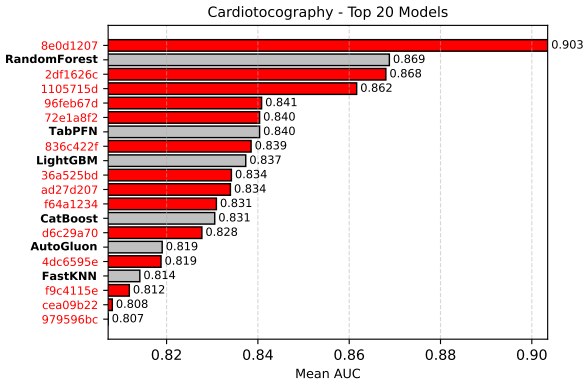


Figure 36: Cardiotocography

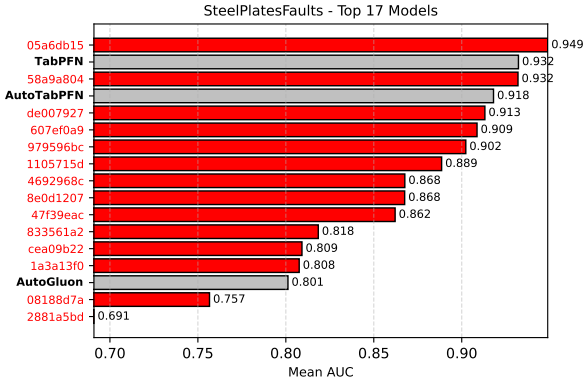


Figure 37: Steelplatesfaults

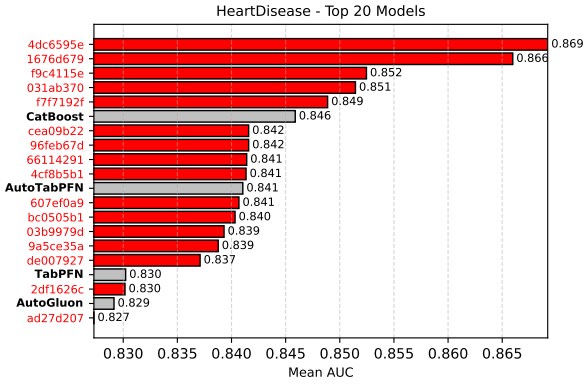


Figure 38: Heartdisease

how this self-improving system can automatically optimize its own architectures, hyperparameters, and even the underlying optimization algorithms—drastically reducing the reliance on human expertise and manual experimentation [2, 18]. In the case of generative machine learning we see new innovations in scaling, feature expansion, feature reduction, and in the fundemental math the creates most algorithms. Some of our most performant generative machine learning models are driven with new-insights that human AI researchers would not consider given our bias. The fences are down with innovation in AI.

This is still a draft, but my hope is the community will begin to believe that all algorithms invented by humans should be examined. Also, this approach is not limited to optimizers or binary classifiers, it can be extended to any algorithm that can be assigned a benchmark. Early work has demonstrated that this approach works for rebuilding

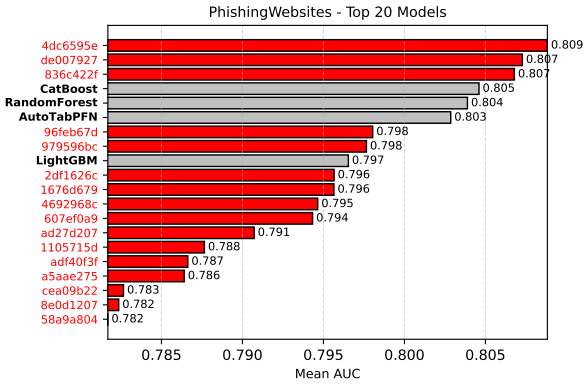


Figure 39: Phishingwebsites

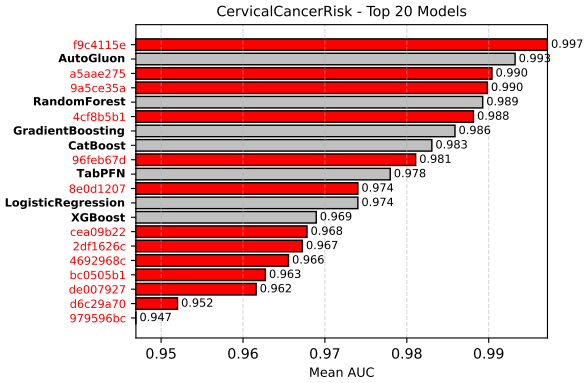


Figure 40: Cervicalcancerrisk

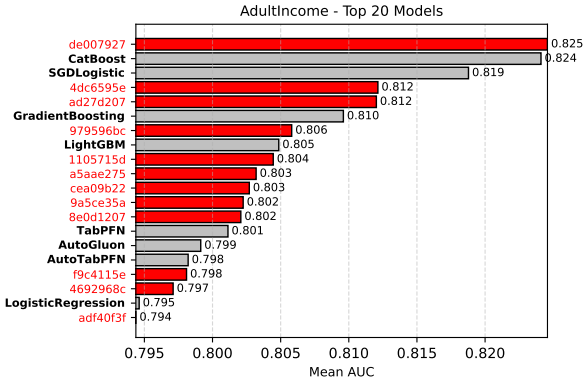


Figure 41: Adultincome

LLM algorithms from the ground up, where transformers are no-longer required. This research, with generative LLMs where the true 'infinite recursion' begins requires significant compute to iterate and test new LLM designs on GPUS. We also have trading performance we plan to include as well when the paper is ready for prime time.

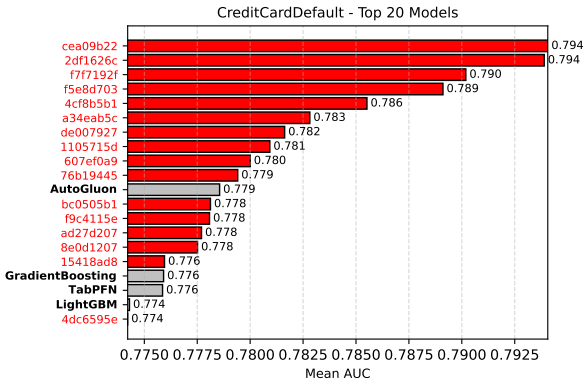


Figure 42: Creditcarddefault

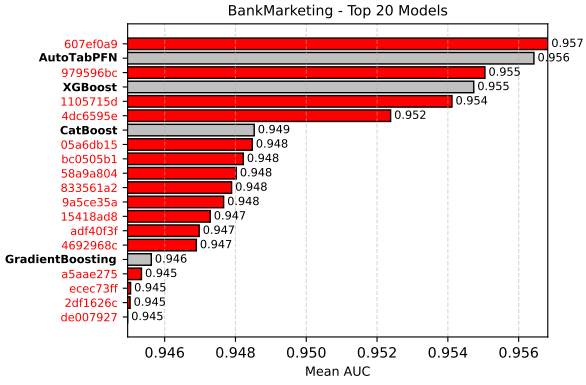


Figure 43: Bankmarketing

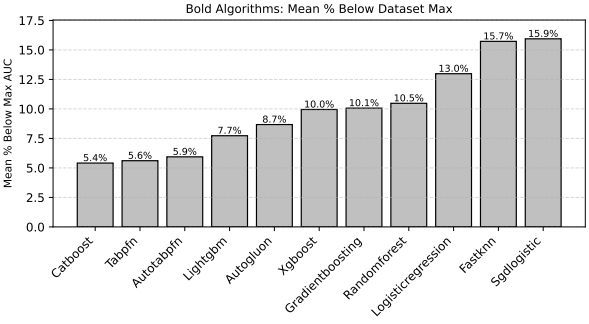


Figure 44: Bold Algorithms: Mean % Below Dataset Max Across All Datasets

A Appendix

B Generative Algorithm Prompt

This shows the detailed required to produce competitive algorithms as a static agent without adaptive systems at play, without this no amount of random function generation will lead to meaningful innovation for optimization test cases.

PROMPT START
Your task is to construct a highly intricate and mathematically rich 2D optimization function $f(x, y)$ using a single-line Python/NumPy expression that presents a rigorous and structurally complex challenge for numerical optimization techniques.
This function will be evaluated on the domain: $x, y \in [-5, 5]$, and it must remain continuous, well-defined, and real-valued across this entire region.
Mathematical Design Considerations The function should exhibit advanced structural characteristics suitable for evaluating modern optimization methods. It must incorporate the following principles:
1. Rich Multimodal Landscape ● The function should contain multiple local minima and maxima across the domain. ● Depth and spacing of these extrema should vary to increase search complexity. ● Introduce interaction between variables to prevent simple separability.
2. Variability in Gradient Behavior ● The function should have non-uniform gradients, including regions of steep slopes and areas of smooth curvature. ● Small perturbations in (x, y) should result in unpredictable shifts in function value. ● Avoid simple quadratic forms—favor intricate, layered mathematical interactions.
3. Multi-Scale Oscillatory Behavior ● Embed both low- and high-frequency periodic components to create a complex response landscape. ● Ensure nested and composite trigonometric-exponential forms, such as: $\sin(x^2 + y^2)$, $\cos(\exp(x \cdot y))$, $\sin(1/x)$ (for $x \neq 0$) ● These elements should interfere in a nonlinear manner, further increasing search difficulty.
4. Broad Output Range and High Variability ● Function values should span a wide numerical range, ensuring large differences in scale. ● Sharp transitions should be strategically placed to require advanced solver adaptability. ● Flat regions should be interspersed with steep ridges, creating numerical instability.
5. Complex Global Minimum Structure ● The function's true global minimum should be difficult to identify. ● Avoid obvious placements—distort symmetry or shift the optimum location non-trivially. ● Implement layered basin structures where local minima obscure the true solution.
6. Non-Separable Cross-Term Interactions ● Ensure x and y interact nonlinearly, preventing separable optimization. ● Examples of desirable complexity: $\sin(xy)$, $\exp(-x^2 y^2)$, $\log(|x + y|)$ ● These elements should disrupt simple descent paths, requiring multi-dimensional exploration.
7. Hierarchical and Layered Function Composition ● Use composite expressions, where one function feeds into another (e.g., $\sin(\exp(x^2 - y^2))$). ● The function should contain both coarse-grained and fine-grained structures, forcing solvers to operate at multiple scales. ● Ensure no single mathematical structure dominates—instead, create hybrid landscapes.
Mathematical Formulation Requirements
Valid NumPy Expression (Single Line Only) ● The function must be returned as a single, continuous string with no additional characters. ● No function definitions (def, lambda), loops, conditionals, or additional logic.
Allowed Variables ● Only x and y (lowercase) may be used.
Allowed Operations ● Arithmetic: +, -, *, /, ** ● NumPy functions: np.sin, np.cos, np.exp, np.tanh, np.sqrt, np.abs, np.log ● Constants: np.pi, np.e
Strict Mathematical Constraints ● Ensure function outputs are always real-valued for all $(x, y) \in [-5, 5]^2$. ● Handle potential negative arguments inside logarithmic terms (use np.abs() where necessary). ● Denominators must never be zero over the defined domain.
Benchmark Function Analysis Your function should incorporate key complexity elements observed in well-known mathematical benchmarks:

BentCigar	Highly anisotropic landscape with extreme directional scaling.
Bohachevsky	Multi-frequency trigonometric interference creating local extrema.
Powell2D	Deep, narrow valleys requiring precise numerical handling.
Dixon-Price	Strong cross-variable interactions requiring coordinated search.
Zakharov	Large polynomial growth that conceals a hidden minimum.
Rastrigin	Dense high-frequency oscillations producing misleading search patterns.
Himmelblau	Multiple widely separated minima requiring extensive exploration.
Levy	Regions of steep variation combined with flat misleading regions.

To surpass these test cases, your function should: ● Blend multiple frequency scales to create a hierarchical difficulty structure. ● Combine exponentials, trigonometric interactions, and polynomial distortions for maximal complexity. ● Introduce variable coupling effects that prevent straightforward gradient-based descent.
Example of Insufficient Complexity The following functions are too simple and will not meet the necessary difficulty criteria:

```
np.sin(x)*np.cos(y) + np.exp(-(x**2 + y**2))
np.square(x) + 2*np.square(y) - 0.3*np.cos(3*np.pi*x) - 0.4*np.cos(4*np.pi*y) + 0.7
```

These fail due to: ● Simple separability in variables. ● Lack of deeply nested interactions. ● Predictable structure with clear convexity or symmetry. ● No interference between hierarchical mathematical layers.
Function Construction Guidelines Your function must: ● Encourage broad and fine-scale exploration through multi-resolution structuring. ● Generate abrupt transitions and variable curvature regions to demand adaptive solver strategies. ● Contain asymmetric perturbations preventing simple direct descent. ● Remain fully real-valued and continuous over the defined domain.
EXPECTED OUTPUT FORMAT (STRICT REQUIREMENT) The function must be returned as a single valid Python string in the following format:

```
"np.sin(...) + np.cos(...) * np.exp(...) - np.log(...) + ... etc."
```

● No newlines. ● No extra spaces. ● No explanations, comments, or additional formatting. ● The entire expression must fit within 3-4 lines of standard document text. ● The function must be executable directly in NumPy without modification.
CAREFULLY study your past failed expressions and their scores:{history} we are wanting to MAXIMIZE these scores
Final Considerations ● Your function should be as structurally complex as possible while maintaining numerical feasibility. ● It must contain unexpected mathematical interactions that challenge solvers beyond existing benchmarks. ● Ensure that no standard numerical method can trivially converge to the solution. ● This function will contribute to the improvement of optimization algorithms by providing a rigorous evaluation landscape. ● Return only the function string, formatted exactly as specified.
ADDITIONAL IMPORTANT INSTRUCTIONS ● Return your function *without* extra quotes. For instance, if your function is:

```
np.sin(x) + np.cos(y)*np.exp(-(x**2+y**2)) - np.log(np.abs(x)+1e-6)
```

do NOT wrap it as:

```
"np.sin(x) + np.cos(y)*..."
```

Simply return the raw expression on one line with no additional quotation marks. ● Do not insert newline characters. ● Do not add disclaimers or any other text.

Algorithm 4 Hybrid Quantum-Classical Optimization (QCO)

Require: Population size N, mutation factor F, crossover rate CR, initial quantum field Q, decay factor δ, tunneling probability T, maximum evaluations M, domain $[a, b]$, objective function f

Initialize population $\{\mathbf{x}_i\}_{i=1}^{N}$ uniformly in $[a, b]^2$
for $i = 1, \ldots, N$ **do**
    Compute fitness $f_i \leftarrow f(\mathbf{x}_i)$
end for
Set best solution $(\mathbf{x}^*, f^*) \leftarrow \arg\min_i f_i$
while total evaluations $< M$ **do**
    Sort population by fitness and define elite set

$$E = \{\mathbf{x}_i \mid \text{top } \max(2, \lfloor N/3 \rfloor) \text{ individuals}\}$$

    for each individual $i = 1, \ldots, N$ **do**
        DE Mutation and Crossover:
        Randomly select distinct indices $a, b, c \neq i$
        Compute mutant:

$$\mathbf{v} \leftarrow \mathbf{x}_a + F\,(\mathbf{x}_b - \mathbf{x}_c)$$

        Generate trial vector $\mathbf{u}$ by replacing each component of $\mathbf{x}_i$ with the corresponding component from $\mathbf{v}$ with probability CR
        Evaluate $f(\mathbf{u})$
        if $f(\mathbf{u}) < f(\mathbf{x}_i)$ **then**
            Update $\mathbf{x}_i \leftarrow \mathbf{u}$ and $f_i \leftarrow f(\mathbf{u})$
            if $f(\mathbf{u}) < f^*$ **then**
                Update best solution $(\mathbf{x}^*, f^*) \leftarrow (\mathbf{u}, f(\mathbf{u}))$
            end if
        end if
    end for
    for each individual $i = 1, \ldots, N$ **do**
        if with probability Q **then**
            Quantum Superposition Sampling:
            Randomly select $\mathbf{e} \in E$
            Generate candidate

$$\mathbf{q} \leftarrow \mathbf{e} + \mathcal{N}(0, \sigma), \quad \sigma = 0.1$$

            Evaluate $f(\mathbf{q})$
            if $f(\mathbf{q}) < f(\mathbf{x}_i)$ **then**
                Update $\mathbf{x}_i \leftarrow \mathbf{q}$ and $f_i \leftarrow f(\mathbf{q})$
                if $f(\mathbf{q}) < f^*$ **then**
                    Update best solution $(\mathbf{x}^*, f^*) \leftarrow (\mathbf{q}, f(\mathbf{q}))$
                end if
            end if
        end if
    end for
    for each individual $i = 1, \ldots, N$ **do**
        if with probability T **then**
            Local Tunneling:
            Generate candidate

$$\mathbf{t} \leftarrow \mathbf{x}^* + U(-\Delta, \Delta), \quad \Delta = 0.5$$

            Evaluate $f(\mathbf{t})$
            if $f(\mathbf{t}) < f(\mathbf{x}_i)$ **then**
                Update $\mathbf{x}_i \leftarrow \mathbf{t}$ and $f_i \leftarrow f(\mathbf{t})$
                if $f(\mathbf{t}) < f^*$ **then**
                    Update best solution $(\mathbf{x}^*, f^*) \leftarrow (\mathbf{t}, f(\mathbf{t}))$
                end if
            end if
        end if
    end for
    Decay quantum field: $Q \leftarrow Q \cdot \delta$
end while
return best solution $(\mathbf{x}^*, f^*)$

References

[1] David H. Ackley. *A Connectionist Machine for Genetic Hillclimbing*. Kluwer Academic Publishers, 1987.

[2] T. B. Brown et al. "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. 2020.

[3] Thomas H. Cormen et al. *Introduction to Algorithms*. A comprehensive textbook covering a wide range of algorithms and their analyses. MIT Press, 2009.

[4] Euclid. *Euclid's Elements*. One of the most influential works in the history of mathematics, laying the foundation for algorithmic thinking. 300 BCE.

[5] M. Feurer et al. "Efficient and Robust Automated Machine Learning". In: *Advances in Neural Information Processing Systems*. 2015, pp. 2962–2970.

[6] C. Finn, P. Abbeel, and S. Levine. "Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks". In: *Proceedings of the 34th International Conference on Machine Learning*. 2017, pp. 1126–1135.

[7] Ian Goodfellow et al. "Generative Adversarial Nets". In: *Advances in Neural Information Processing Systems*. 2014, pp. 2672–2680.

[8] Andreas Griewank. "Generalized descents for global optimization". In: *Journal of Optimization Theory and Applications* 34 (1981), pp. 11–39.

[9] Jonathan Ho, Ajay Jain, and Pieter Abbeel. "Denoising Diffusion Probabilistic Models". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 6840–6851.

[10] Muhammad ibn Mūsā Al-Khwarizmi. *The Compendious Book on Calculation by Completion and Balancing*. A seminal work in algebra whose methodologies gave rise to the term "algorithm". 820.

[11] Donald E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. A classic reference that covers the principles of algorithm design and analysis. Addison-Wesley, 1997.

[12] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.

[13] A. Krizhevsky, I. Sutskever, and G. E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems*. 2012, pp. 1097–1105.

[14] Y. LeCun, Y. Bengio, and G. Hinton. "Deep Learning". In: *Nature* 521.7553 (2015), pp. 436–444.

[15] Y. LeCun et al. "Gradient-Based Learning Applied to Document Recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[16] Zbigniew Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, 1992.

[17] V. Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533.

[18] OpenAI. *GPT-4 Technical Report*. Preprint available at `https://arxiv.org/abs/2303.08774`. 2023.

[19] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[20] Liudmila Prokhorenkova et al. "CatBoost: unbiased boosting with categorical features". In: *Advances in Neural Information Processing Systems*. 2018, pp. 6639–6649.

[21] J. R. Quinlan. "Induction of Decision Trees". In: *Machine Learning* 1.1 (1986), pp. 81–106.

[22] L. A. Rastrigin. "Systems of extremal control". In: *Proceedings of the USSR Academy of Sciences*. 1974, pp. 161–172.

[23] F. Rosenblatt. "The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain". In: *Psychological Review* 65.6 (1958), pp. 386–408.

[24] Howard H. Rosenbrock. "An automatic method for finding the greatest or least value of a function". In: *The Computer Journal* 3.3 (1960), pp. 175–184.

[25] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. "Learning Representations by Back-propagating Errors". In: *Nature* 323.6088 (1986), pp. 533–536.

[26] Hans-Paul Schwefel. "Evolution strategies: A family of non-linear optimization techniques based on imitating some principles of organic evolution". In: *Annals of the 1st Int. Conf. Cybernetics*. 1983, pp. 160–165.

[27] Peter W. Shor. "Algorithms for Quantum Computation: Discrete Logarithms and Factoring". In: *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. A groundbreaking paper that introduced quantum algorithms with the potential to revolutionize computational complexity. IEEE. 1994, pp. 124–134.

[28] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529 (2016), pp. 484–489.

[29] David Silver et al. "Mastering the Game of Go without Human Knowledge". In: *Nature* 550 (2017), pp. 354–359.

[30] Alan M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society* 42 (1936). A foundational paper in the theory of computation and the limits of algorithmic processes., pp. 230–265.

[31] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, 1995.

[32] A. Vaswani et al. “Attention is All You Need”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 5998–6008.

[33] Vijay V. Vazirani. *Approximation Algorithms*. A key resource on the design and analysis of approximation algorithms for complex problems. Springer, 2001.